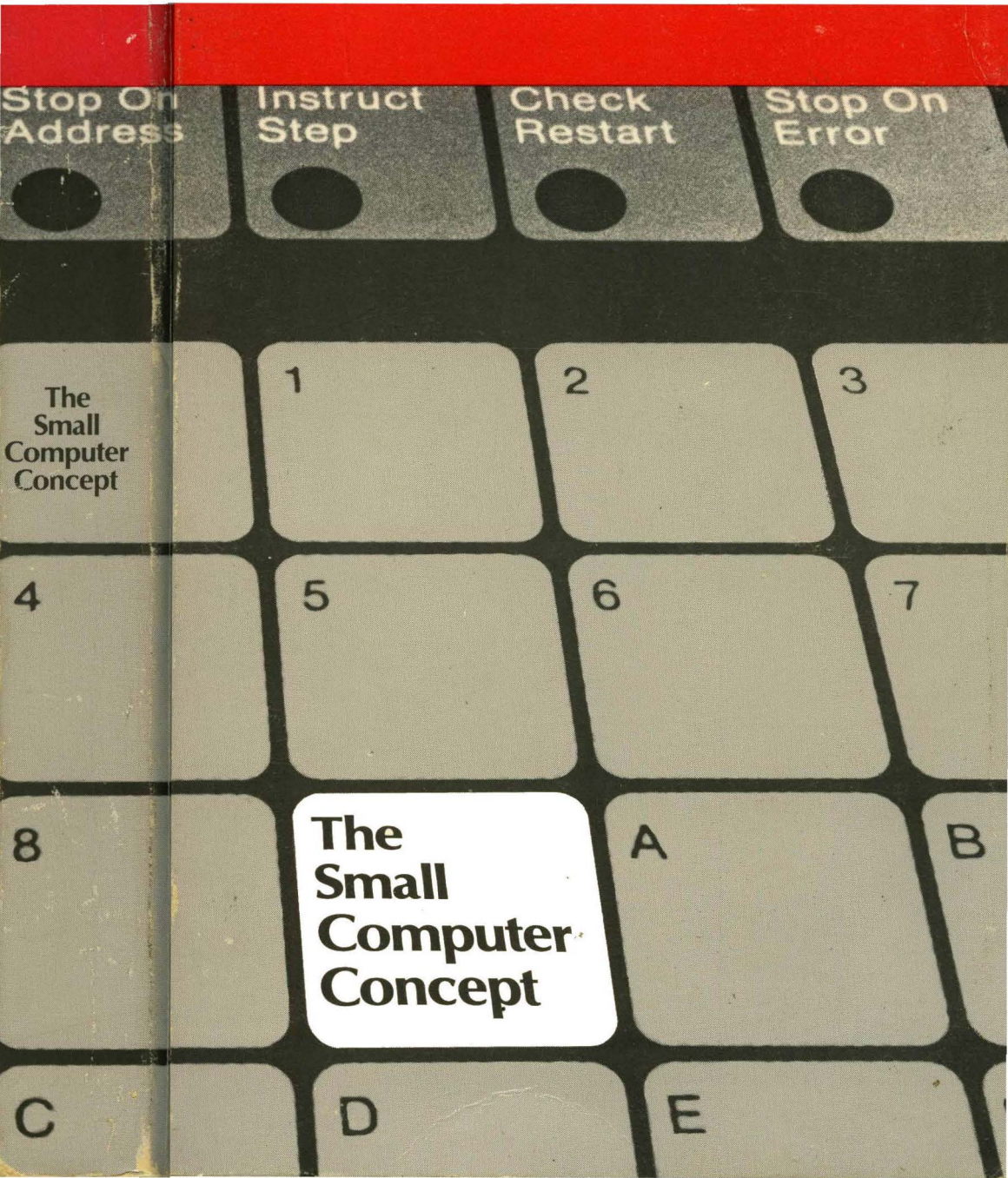
IBM

Series/1

IBM

Series/1



The Small Computer Concept

IBM Series/1

The Small Computer Concept

by

James D. Schoeffler

Professor and Chairman
Department of Computer and Information Science
Cleveland State University
Cleveland, Ohio

International Business Machines Corporation
General Systems Division
Atlanta, Georgia

Library of Congress Catalog Card Number: 78-61315

IBM Order Number: SH30-0237

Additional copies of this book can be obtained from local IBM Branch Offices, using the IBM Order Number.

Comments concerning this publication should be addressed to International Business Machines Corporation, General Systems Division, Technical Publications, Dept. 796, P.O. Box 2150, Atlanta, Georgia 30301.
IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information supplied.

First edition published August, 1978

Printed in the United States of America

Preface

The trend toward small computers and distributed systems for a wide variety of applications has been termed a revolution by some. Although the impressive decrease in the cost of these systems has certainly made this trend feasible, the driving force has been the changing nature of computer use. Applications today tend to be much more interactive and online and less suitable for batch processing. Because such applications are critical to the operation of business or industry, they place stringent demands on system hardware, application and system software, and system maintenance.

Consequently, the choice of a small computer system is not totally determined by its price. Unfortunately, too many small computer systems have been designed to minimize price rather than to meet such critical specifications. Hence, users have often been frustrated in their attempts to utilize the benefits of small computers and distributed applications.

After being exposed to the designers and planners of the Series/1 following its introduction by IBM in November 1976, I was enthusiastic about writing this book. I believe that the Series/1 was expressly designed to meet the kinds of critical applications indicated above. I view it as an "integrated system" in the sense that the hardware, software, and maintenance concepts all were designed from the beginning to work closely together so that critical applications can be realized. Furthermore, the system design is such that non-IBM devices can be included in the system without sacrificing performance, use of system software, reliability, or availability. This aspect of small computer applications is very important to the original equipment manufacturers designing application systems around such a computer.

For the above reasons, I have organized the book around this integrated hardware, software, and maintenance concept. The book is intended for users who wish to critically evaluate the IBM Series/1 for their applications and want to know what it does, how it does it, and why it does it that way. The book is not a handbook or reproduction of the various system reference manuals. It is not an attempt to describe every processor, software product, or device in great detail.

Rather, those items which I considered important to the overall integration of the product into successful applications have been included.

Throughout this project, I have been aided by useful suggestions from a variety of IBM people, most of whose names I do not even know. However, three people in particular contributed significantly: Mr. Michael I. Davis, one of the principal architects of the Series/1; Mr. Harry J. Dewhurst of the marketing group; and Mr. Charles E. Snyder, my editor for the book. In my many discussions of the Series/1, its objectives, its realization, and its future, I have been impressed with their knowledge and insight of small computers and grateful for the opportunity of knowing and working with them.

James D. Schoeffler July, 1978

Contents

Figures

Tables

Introduction to the Small Computer Concept

Industry, banking, manufacturing, marketing, health care, and many other enterprises have achieved increased productivity in an impressive variety of applications by using small computer systems. These applications have been and are successful because users have implemented the total, small computer *system* concept instead of relying on the operation of the small computer *by itself.*

A system is a set of closely interacting subsystems which, in turn, have component parts. It is important, however, to differentiate between the use of the word *system* when it is applied to small computers and when it is used in other contexts. Typically, to create a system in small computer applications, end users and third party vendors must integrate processors, peripherals, software systems, and application software.

Conceptually, we should emphasize that the "tools" or components made available from a variety of sources must be integrated into a system that meets the application needs of the individual user. Frequently, manufacturers design these components completely independent of one another; consequently, integrating them into a single system is neither a straightforward nor a simple task. This chapter discusses four small computer applications and will serve as an introduction to computers' systems.

Users, whether they are original equipment manufacturers imbedding a small computer into their products or are end users, need computer components that can be integrated as successfully as if each one were designed by the same vendor to function solely for the individual user's application. Consequently, small computer applications are particularly sensitive to three constituents:

1. An integrated system of hardware components
2. An integrated system of software components
3. An integrated system of maintenance and support

An integrated system, in this context, means that the design of each particular part of the hardware, software, maintenance equipment, or maintenance procedures recognizes that users will later integrate these discrete components into their application systems. To enable each user to do this, there must exist a comprehensive and detailed overall system organization or "architecture" for hardware, software, and maintenance.

Small Computer Application Needs

The term "architecture" describes the overall organization and discipline of the interconnections among the integrated system of hardware and software components and their maintenance. A modern architecture insures that the user of these systems can incorporate future developments in memories, auxiliary storage devices, distributed systems, and other peripherals without a complete system hardware and software redesign. Similarly, an integrated system of software components—including, for example, operating system software and language preparation facilities—is essential if users are to realize the economic benefits of their systems. Moreover, the software system must utilize the architecture of the hardware to enable both components to work together as a system.

Operating systems must anticipate the variety of present and future application demands so the user does not have to maintain an excessive number of systems over an extended

period of time. Reliable and serviceable hardware not only gives a long mean-time-between-failures but also facilitates error diagnosis so repairs can be accomplished quickly when failures occur. Because small computers often involve a mixture of special attachments, extensive self-diagnosis is especially important to pinpoint the source of difficulty to a printed circuit card or device. Firm vendor commitment to this diagnostic concept through worldwide maintenance of hardware and software for the expected lifetime(s) of the system(s) is necessary if users, in turn, are to make firm commitments to small computers. Vendor commitments should include user access to trained analysts and engineers when subtle problems arise in the hardware or software during development of a new application. Only when all of these components are present is a user in a position to attain an economical use of the small computer over an extended period of time.

The IBM Series/1 has been expressly designed to serve these user requirements. It is a family of small computer hardware components, integrated with an extensive range of software systems and self-diagnostic capabilities, and backed up by a strong maintenance force. The IBM Series/1 is a general-purpose family intended to serve many application areas.

The name *Series/1* indicates that the hardware, software, and maintenance products can be used as components or "tools" to build small computer application systems (Figure 1). These components are part of a "system" because Series/1 has an overall hardware, software, and maintenance architecture and IBM has designed each component to function specifically within this structure. Furthermore, the overall architecture has been composed to permit non-IBM hardware and software to be fully and successfully integrated into the final system. This book emphasizes the overall system concept because successful implementation of this concept is crucially important to the success of original equipment manufacturers, system integrators, and end users of small computer systems.

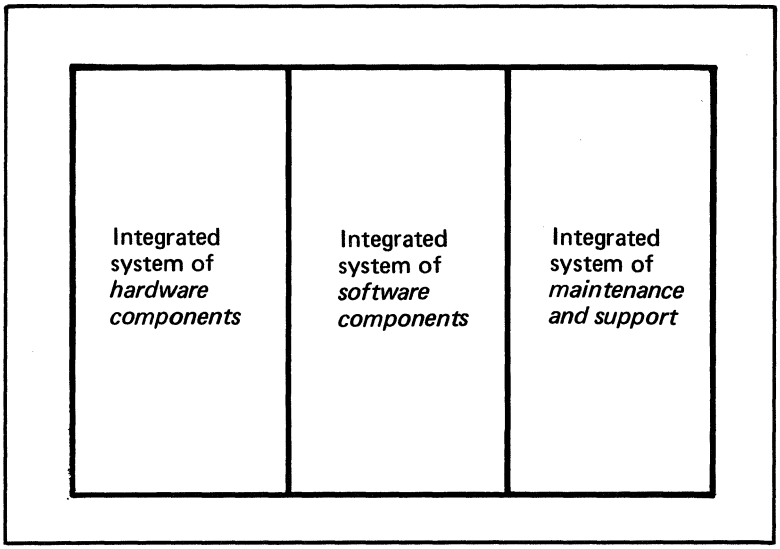


Figure 1. Essential ingredients of a small computer system

This chapter examines several small computer application areas, identifying their application-imposed needs in hardware, software, and support. These applications are presented here in the way that they are currently implemented in the industry and not necessarily as they might be realized in the IBM Series/1 architecture. Chapter 2 introduces the specific IBM Series/1 hardware and software architectures which answer these identified needs.

Four diverse but very common small computer applications are:

1. The multifunction terminal
2. The communications' concentrator
3. The front-end processor
4. Data acquisition and control

These examples illustrate the broad range of applications for which the small computer is a suitable solution. By examining typical hardware configurations and software implementations, it is possible to abstract the hardware, software, and maintenance requirements for each of these applications.

Terminology usage varies considerably within the industry. In this chapter, conventional terminology (e.g. Direct Memory Access [DMA]) is used and related to the standard Series/1 terminology (e.g., processor I/O channel). The remainder of this book and all IBM documentation use the standard Series/1 terminology.

The Multifunction Terminal Application

Small computers have opened many alternatives to business data processing. Among the most common and successful methods of data handling are:

- Key entry—the process of entering data; converting human-oriented documents into machine-readable media; formatting, editing, and compacting data; and finally—entering the data directly or indirectly into a computer through magnetic tape, diskette, or disk intermediate storage.
- Remote job entry—a remote terminal controls peripheral devices such as card, diskette, or tape readers and line printers. The terminal accepts jobs through the input devices, transmits them to a remote computer, receives the output from the computer, and produces reports locally. Remote job entry terminals permit quick access to centralized computer systems even when the users are not physically proximate.
- Transaction processing on a local data base—a small computer system maintains a local data base which the user frequently accesses and updates. Locally, the system executes special application programs. Access to a host computer system for either data storage retrieval or more extensive computational tasks is a characteristic of this terminal.

The simplest data entry application provides a key punch replacement by collecting the characters in one record in the small computer main storage and transmitting it to an output program when the record is complete. A more useful form of data entry permits formatted input; the operator selects

one from a number of predesigned formats. If the number of formats is large, they are stored in auxiliary storage with only the active formats in main storage at any given time. Validity checking of data fields, prompting of operators, and other standard operations can occur with this form of data entry. Similar functions are desirable in transaction processing applications where the type of transaction determines the data needed for that transaction.

Typical applications where the multifunction terminal is desirable include:

Order Processing

- Entering orders
- Inquiring about the status of orders
- Interactive specification of data associated with an order
- Reporting the status of orders and maintenance of the local order data base

Warehouse Inventory Control

- Maintaining the local data base
- Matching orders against inventory
- Communicating with a remote computer to get orders and to report the status of orders
- Interactive inquiry into the inventory status
- Reporting of overall orders and the inventory status

Plant Scheduling and Production Data Collection

- Gathering data from operator terminals about production status
- Maintaining a local data base containing current orders and the status of in-process inventory
- Inquiring about the status of orders, labor, production, and machines
- Rescheduling in response to daily events
- Communicating to remote computers for overall plant control

Similar applications occur in many other industrial and service activities.

Hardware Support

Each of these business data processing applications requires similar hardware devices and software support. There is a logical trend in the data processing industry to combine these functions within the so-called multifunction terminal. The specific number of keyboard/CRT stations, types of peripherals, speeds and numbers of communications' ports and volume-auxiliary storage varies with the specific user; but the application needs can be summarized.

Figure 2 shows the hardware system needed to support the multifunction terminal application. Terminals generally provide keyboard and alphameric display capability but they can be more extensive; for instance, they may provide programmable displays to guide transaction data entering and validation. The system must support data storage devices, especially the simpler ones like diskettes. Communications to the remote or host computer can be at low, medium, or high speeds as the application (and costs) dictate. Consequently, the communications' hardware support must incorporate these requirements. Frequently, special display or output devices are needed.

Since no single computer vendor can supply every type of device for all applications, users must be able to easily attach other vendors' devices. Otherwise, the multifunction terminal would have limited application: manufacturers would have to develop a discrete terminal for each special application—a costly duplication of effort. From the point of view of the terminal supplier, this generality of small computer hardware architecture and support of special devices permits suppliers to add to their product lines economically without incurring a "dead end" design. A general hardware specification for this application can be identified.

> Small computers should have an architecture compatible with both large and small systems.

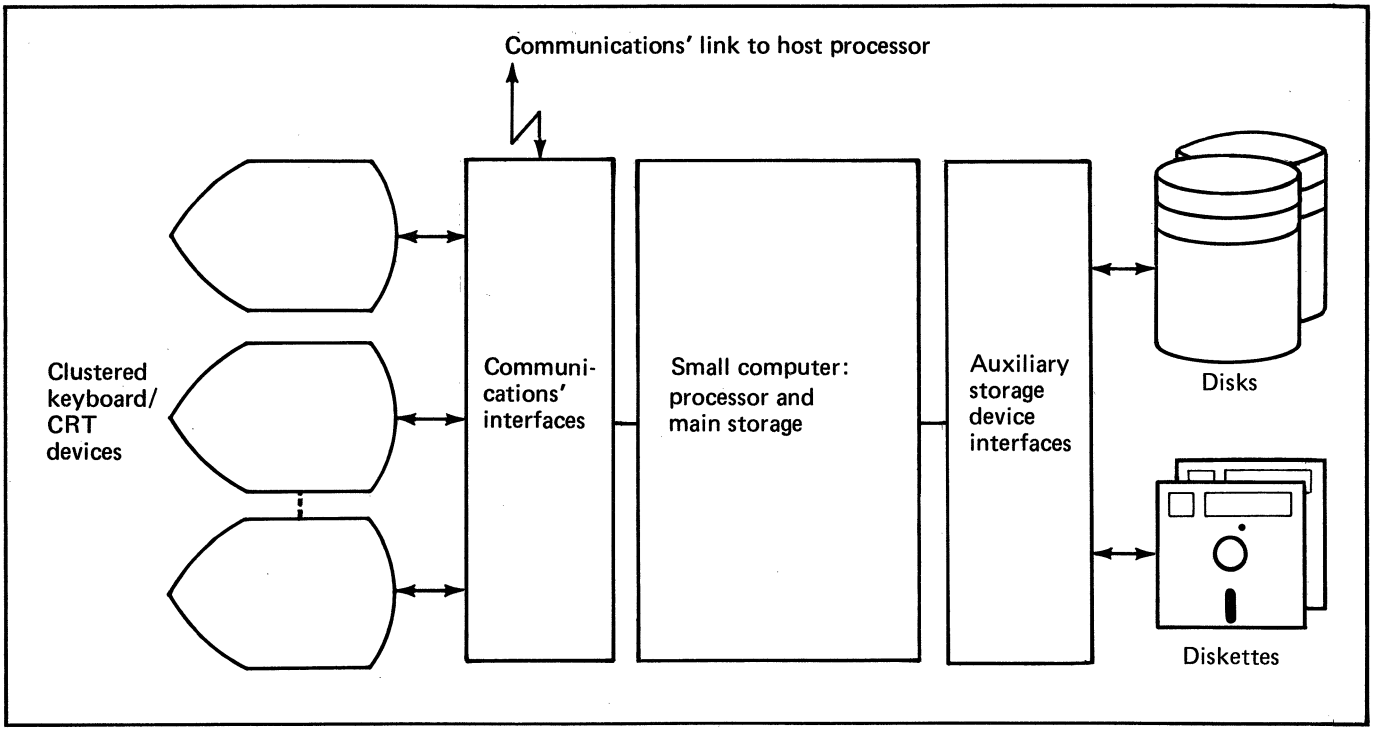
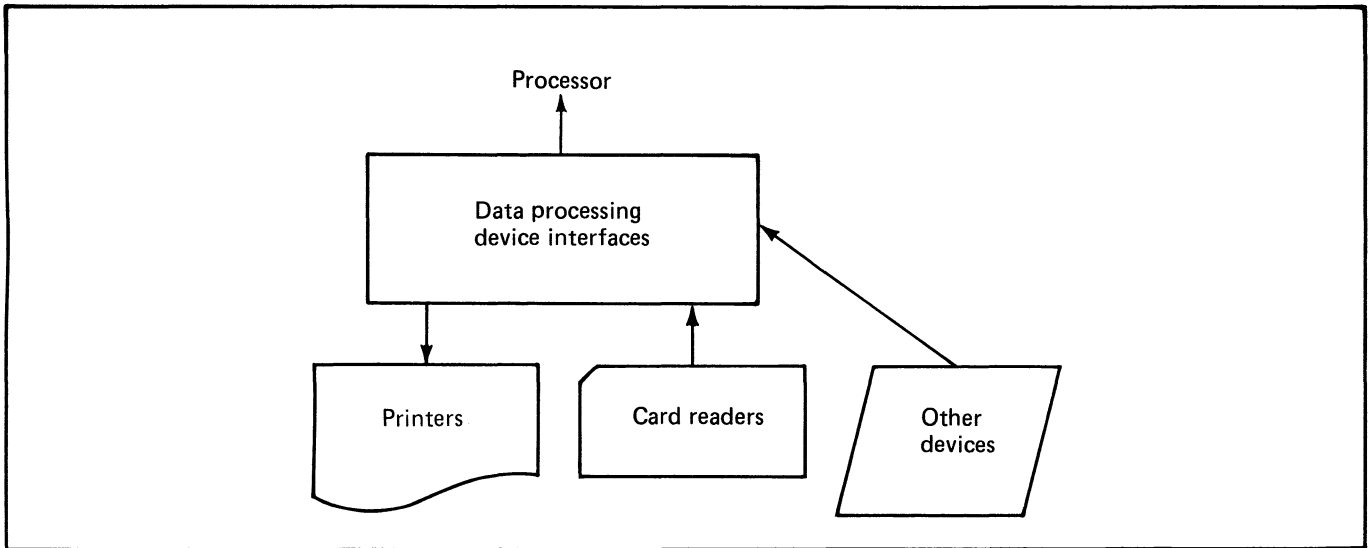


Figure 2. Common configuration for a multifunction terminal application (1 of 2)



The multifunction terminal combines, on a local data base, common business data processing applications like key entry, remote job entry, and transaction processing—which have similar hardware, software, and maintenance needs—into a common terminal with great flexibility and wide applicability.

Figure 2. Common configuration for a multifunction terminal application (2 of 2)

The small computer should have an instruction set capable of supporting both business data processing and communications-oriented applications.

The hardware system should effectively integrate communications' interfaces with the small computer itself.

Hardware and software compatibility are requirements for an economical application. Figure 3 shows a representative organization of the software needed to execute the multifunction terminal application. Certain characteristics of this software—for instance, its definition as a set of programs that interact to perform the application—occur repeatedly and dictate both the types of software needed in a small computer system environment and the types of hardware needed to support that software.

For example, user requirements for terminal peripherals vary widely: an elementary school education application might require a mark-sense card reader; a research laboratory application might need a high-speed printer.

The hardware and software systems must support the user's addition of input/output devices to the system.

Partitioning an application into a set of related and interreacting programs is a good way to design and implement an application. Programs are asynchronous. Typically, they need different kinds of data, devices, and other system resources like error recovery procedures. Breaking up the application into a set of programs structures the application into units of work—each unit performing a given function.

Software Support

The integration of the communications' function into the hardware and software, while it is not as critical in this

application as it is for dedicated communications' applications, is still crucial to the system's maximization of terminal throughput.

Support Needs

The third element of the computer system design shown in Figure 1 is the support needs. Multifunction terminals are often used in office environments where on-site maintenance personnel are not knowledgeable about computers. In such instances, the user commonly contracts service for the computer system and its peripherals from the computer vendor.

After they are fully operational, applications like multifunction terminals become essential to a company's performance. Consequently, it is important that the system be accessible most of the time and, when it fails, that service be readily and promptly available. This service must remain available throughout the extended lifetime of the small computer because it is not economical for users to redesign their application software and buy new hardware for their terminals whenever the computer vendor markets a new generation of hardware or software.

The multifunction terminal application needs are easily matched with small system hardware, software, and maintenance specifications. When these specifications are explicit, they not only permit users to evaluate a particular system relative to their application but also provide a guide as to the best use of the vendor's hardware, software, and support products.

Chapter 2 matches the multifunction terminal application needs with the IBM Series/1.

The Communications' Concentrator Application

Communications involve the transmission and reception of data, through messages of various formats, between devices which may be long distances apart. The small computer is especially attractive for this application because

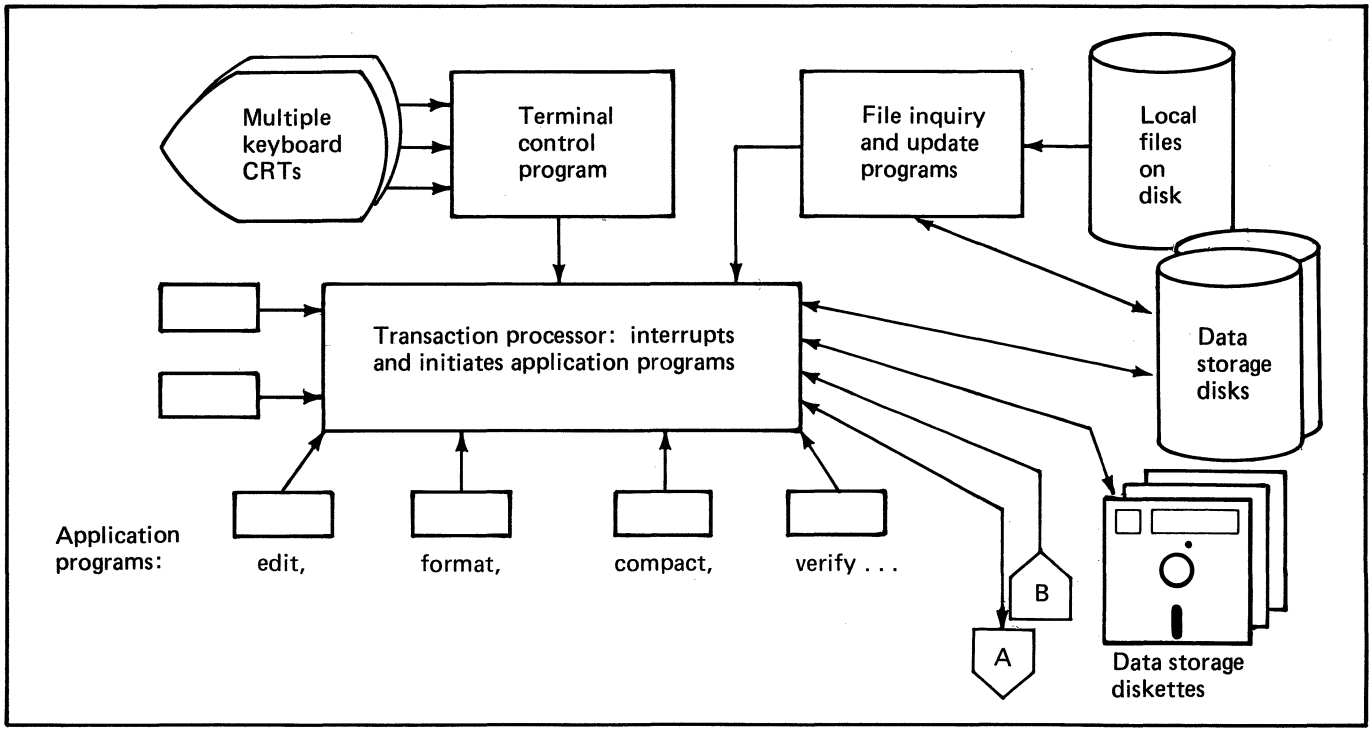


Figure 3. Multiple cooperating programs for the multifunction terminal application (1 of 3)

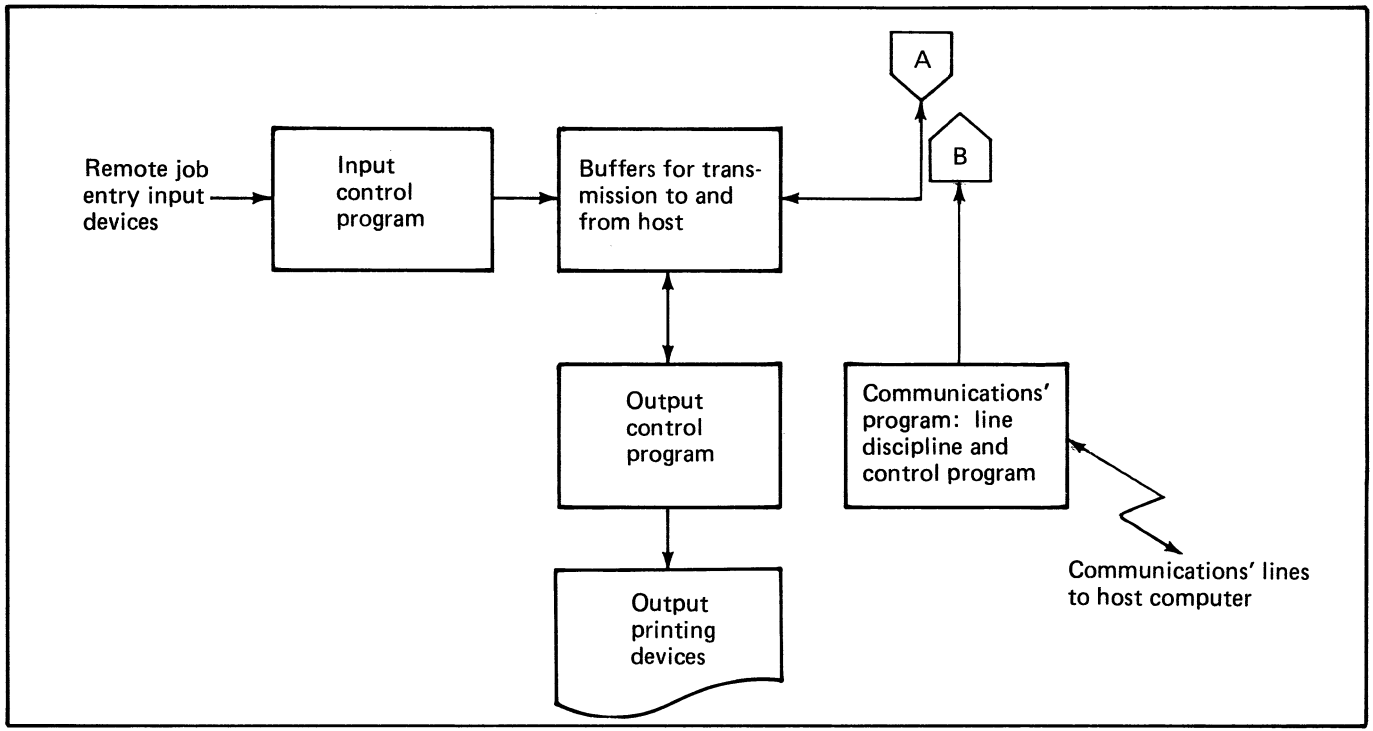


Figure 3. Multiple cooperating programs for the multifunction terminal application (2 of 3)

Many different programs cooperate to support transaction processing:

- Key entry
- Inquiry from keyboard/CRT devices into data bases
- Remote job entry
- Printing of output
- Communication with the host

Software support, through a general purpose operating system, permits the combining of these business data processing applications into a single terminal.

Size scaling for various users involves:

- Addition or deletion of processors
- Changing the size of the files
- Changing types of communications' modes

Size scaling does *not* change the generic application configuration or software organization.

Figure 3. Multiple cooperating programs for the multifunction terminal application (3 of 3)

its programmability permits an interconnection of terminals and devices which have widely differing characteristics—like varying transmission and reception speeds. Furthermore, the small computer system can take advantage of main and auxiliary storage to buffer various communicating devices; this insures that the overall system is not sensitive to different communications' rates or intermittent unavailability of lines or devices.

The Concentrator Function

Frequently, the relatively high rental cost of communications' lines—in many applications accounting for a very significant part of the total teleprocessing system costs—is a major barrier to wide acceptance of data communications. The line costs depend primarily on the type of line used (leased or switched), the distance of the terminal(s) from the computer site, line quality, and bandwidth. Using fewer, high bandwidth lines—a major benefit provided by a concentrator—instead of many, low bandwidth lines can reduce line costs considerably.

The primary function of a concentrator is to consolidate the input from a group of clustered terminals and transmit their combined data at high speed over a single line (or fewer lines) to a remote computer (Figure 4). In the past, special purpose, hardwired, hardware multiplexers have performed the data concentrator functions. Now, the availability of programmable, small computers with concentrators adds new dimensions to, greater flexibility for, and more economy in data processing applications.

Concentrators are actually multiplexers enhanced with a buffer and a processor. They are more complex and more expensive than multiplexers but they can also do more to reduce line costs. For instance, users may program concentrators to perform the following functions:

- Accommodate changes in format, codes, data rates, and communications' procedures
- "Smooth" traffic
- Compress data

- Adapt channel characteristics
- Pre-scan data for unacceptable formats and errors

Hardware Support

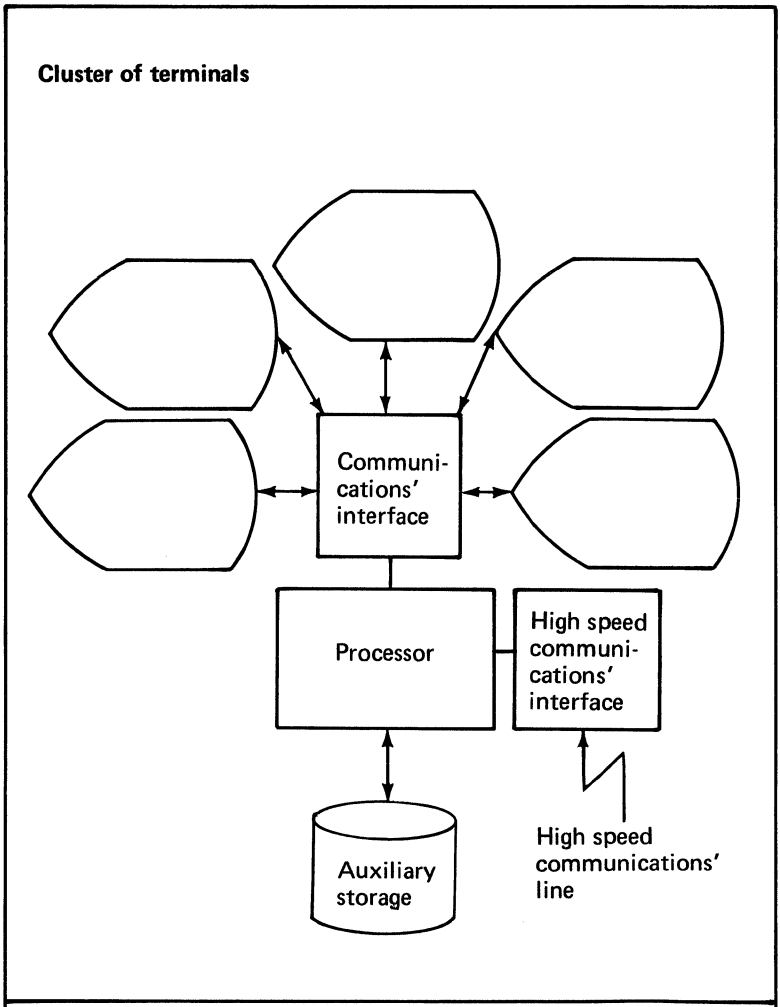
In communications' applications, certain hardware and architectural needs are more critical than they are in other types of applications. For example, there is a critical dependence on efficient communications' interfaces.

> To support the variety of transmission modes currently available, the small computer system must have extensive communications' interfaces; moreover, the communications' hardware design must be efficiently integrated into the system architecture.

This integration requires both a variety of interfaces supporting the standard line speeds and communications' disciplines, and adequate control from the processor so that the system can test the integrity of these interfaces for diagnostic purposes. The interface should do more than simply collect characters and insert them in storage. It should also recognize critical or control characters and—under management of the processor—interrupt where appropriate. Such flexibility simplifies the software and is a good example of a carefully integrated hardware/software system.

Software Support

Figure 5 illustrates the programs necessary for the concentrator application. Note those programs that are concurrent because of the lack of timing control when information comes from terminals or communications' lines. Effective programming is critical because programs manipulate data at the bit and byte level; since the throughput of the system is usually very high, execution speed is a paramount consideration. Generally, all of the hardware and software characteristics previously mentioned in this chapter are apparent in communications' concentrators.



The concentrator consolidates input from a group of clustered terminals and transmits their combined data, at high speed, over a single line (or a few lines) to remote computers.

Users may connect terminals directly or input data through communication lines which are relatively short in length compared to the distance between computers.

Auxiliary storage permits message buffering for terminals.

Figure 4. Concentrator configuration

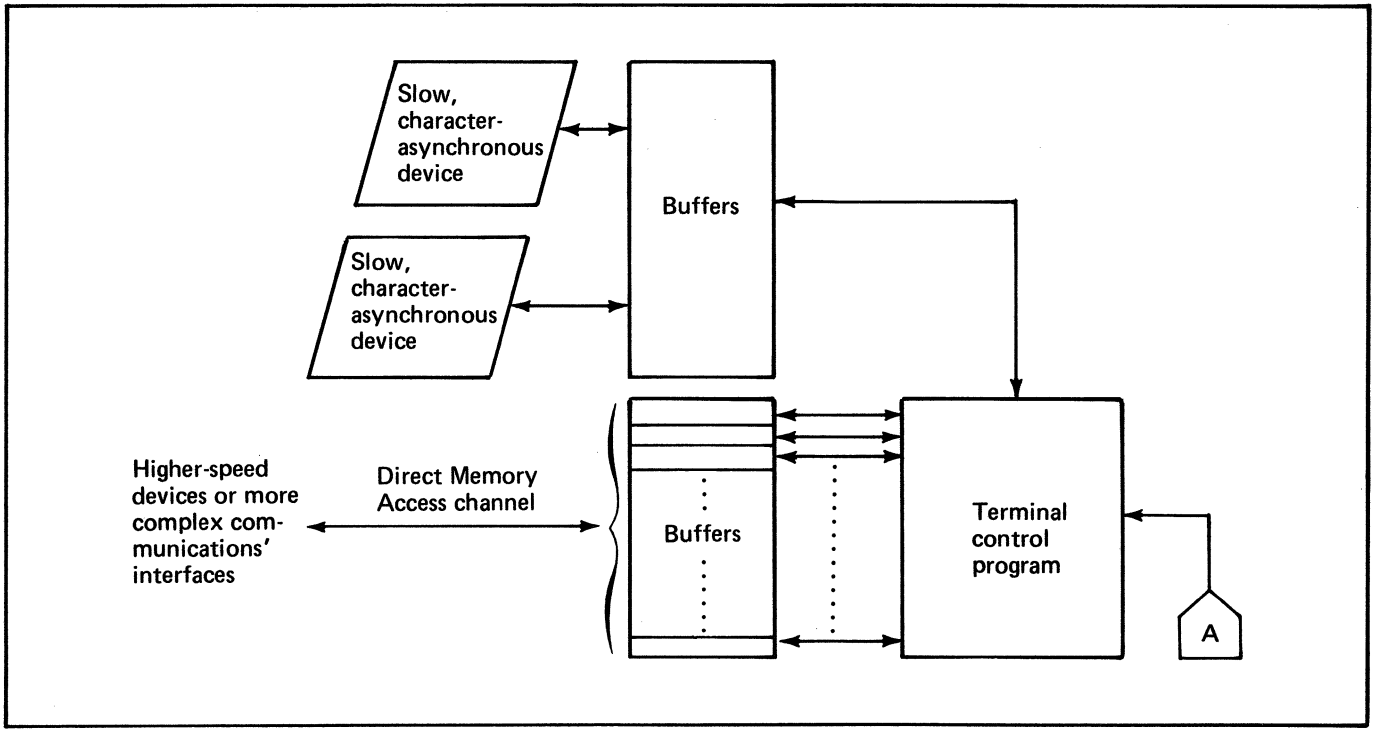


Figure 5. Concentration of communications (1 of 3)

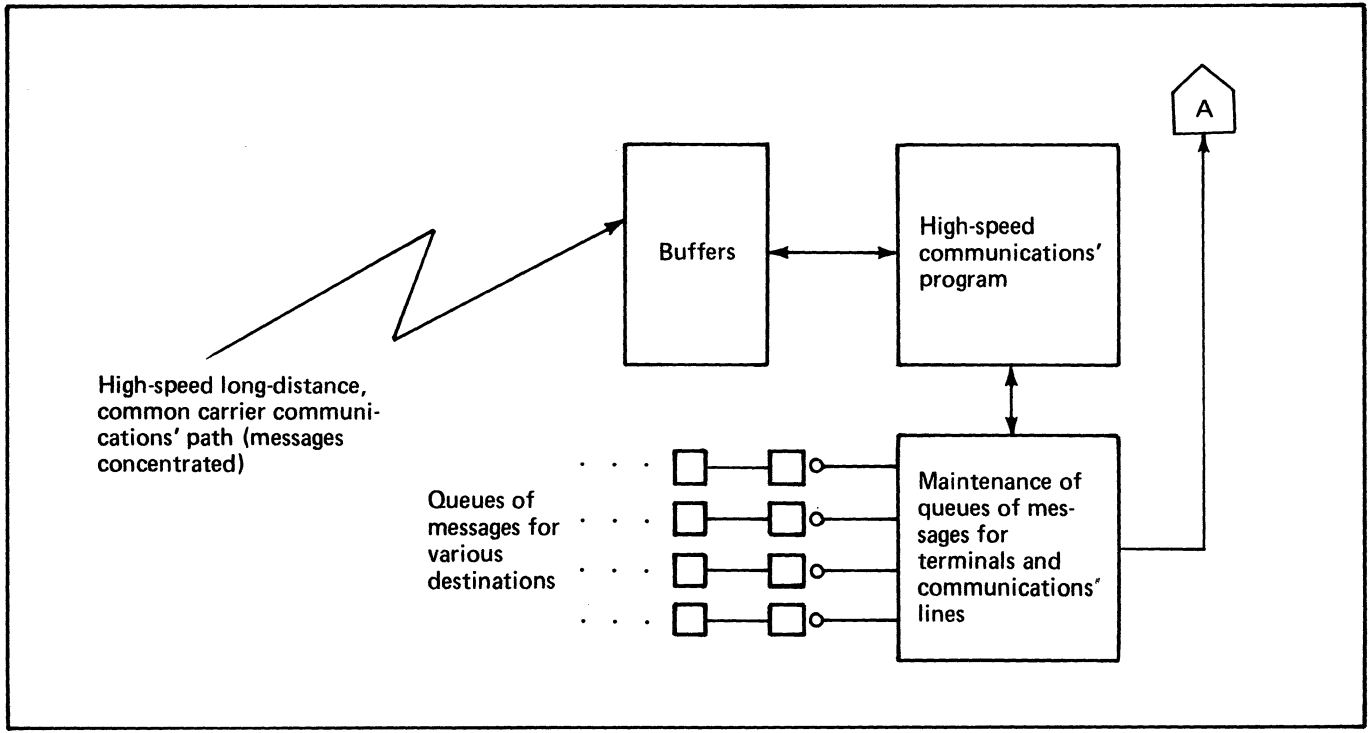


Figure 5. Concentration of communications (2 of 3)

Using the necessary communication modes and speeds, terminal control programs perform all high- and low-speed, input/output operations with local terminals.

Local data involves queues of messages waiting for transmission to a remote site, or waiting for outputting to a local terminal.

Messages are formatted and grouped for high-speed transmission to remote terminals.

More sophisticated interfaces can remove some of the programs: for example, asynchronous device interfaces can input characters through the Direct Memory Access channel and signal via interrupt when a control character is encountered.

Figure 5. Concentration of communications (3 of 3)

At the lowest level of consideration, a communications' attachment must be available to provide an interface to the actual device transmitting and receiving signals (the modem) across the dedicated or common carrier lines to the computer itself.

Most applications use a variety of low-, medium-, and high-speed communications' methods and must utilize interfaces for all of them. Many communications' methods identify special characters and use them for control purposes (e.g., to indicate that the system has received a message correctly, or that a terminal wishes to transmit data). Because the particular control characters vary widely, the user must be able to program the communications' interfaces. Interface identification of such a control character and subsequent action notification to the small computer itself also require programmable interfaces. Software support for communications conditionally involves basic input/output routines whose functions are:

1. To accept characters into an area in storage (a buffer) until either a line is complete or a control character has been received
2. Then, to notify the communications' program, which
   a. Interprets the message
   b. Handles the communications' protocol with the communicating remote device
   c. Finally, activates the appropriate program for which the message is intended

Because the overhead would be intolerable, an application involving many terminals or high rates of transmission cannot interrupt the small computer for each character received. Instead, the communications' interfaces must insert characters into storage directly through Direct Memory Access (DMA) and interrupt only when the system detects a control or critical character.

For example: the system accepts characters transmitted from a remote terminal and inserts them directly into a storage buffer until the system detects the end-of-line character. The computer then receives an interrupt to enable

the communications' program to handle this line and set itself up for the next line to be received.

In this operation, several important needs appear, including:

- Multiple programs cooperating to carry out an application
- Concurrent execution of programs
- Fast response to external events
- Extensive manipulation of information items in bytes and words

The Front-End Processor Application

Closely related to the concentrator application is the use of the small computer to relieve the main application computer (whether large or small) from the critical communications' program. The front-end processor application accomplishes this: essentially, by removing much of the data communications' control function from the central computer.

The user can incorporate complex, computer-network communications' methods into the front-end processor so that the applications become completely independent of the types of communications' terminals, methods, and the specific communications' codes that are used in the application.

Front-end processing is potentially the most important application of communications' processors because it can increase communications' throughput and processing efficiency, save programming, reduce equipment and line costs, and even extend the life of the processing facilities.

Hardware and Software Support

Figure 6 shows a possible configuration for a front-end processor. Flexible and varied communications' interfaces are obviously important. As illustrated, the concentrator and front-end processor organize software in a similar manner.

Both the front-end processor and concentrator applications require software that responds rapidly to interrupts

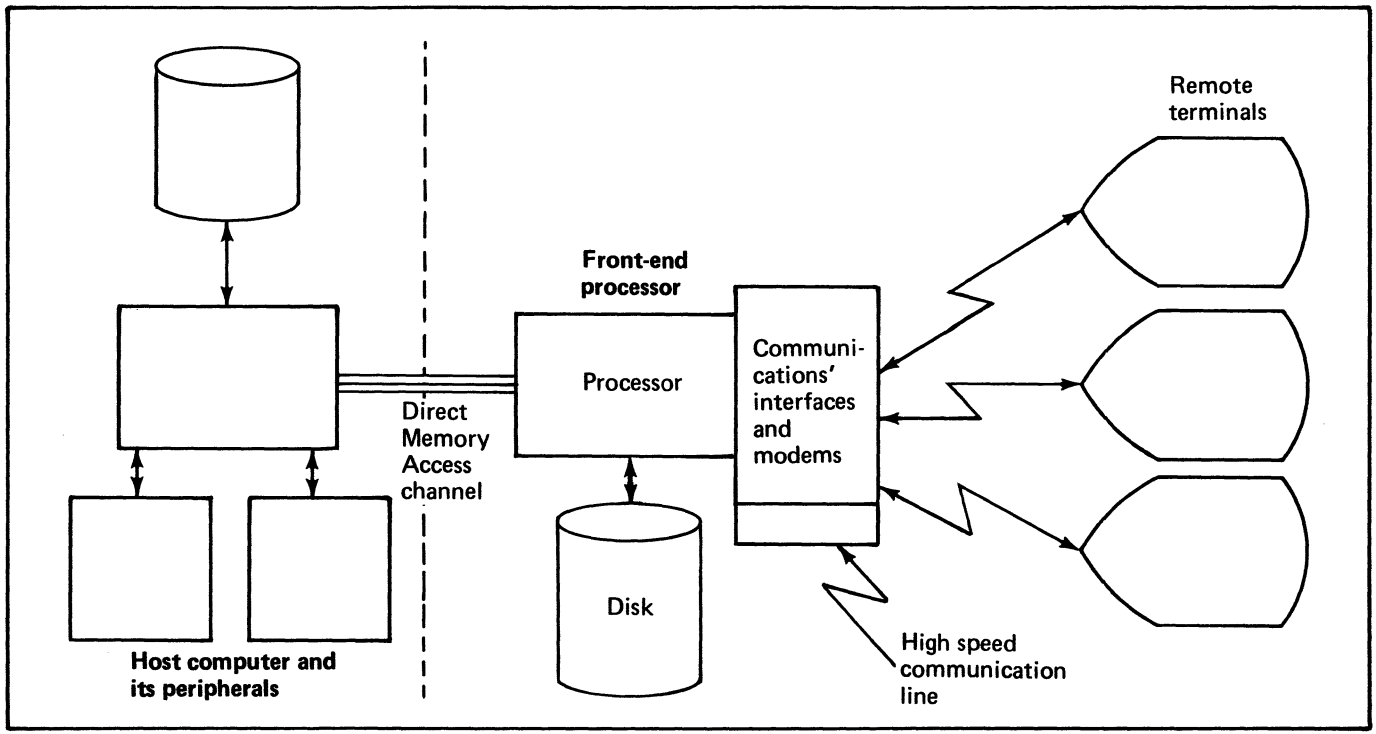


Figure 6. The front-end processor (1 of 2)

Software organization for the front-end processor is similar to that for the concentrator—a set of programs to:

- Control specific classes of terminals
- Perform communications' protocol functions
- Maintain buffers and queues
- Control I/O to the host computer

The front-end processor removes all data communications' control from the host computer and thereby reduces its workload considerably.

The front-end processor incorporates all the required communications' protocol, buffering, error recovery, and formatting of messages.

Communication with the host is through Direct Memory Access channels.

Figure 6. The front-end processor (2 of 2)

and events, and that can switch rapidly from program to program in the application software complex. Because speed and throughput are more critical than generality of operating system features, communications' application requirements for the operating system are very different from those in the multifunction terminal application.

> When it is essential to the application, system software must permit users to customize their systems.

In critical applications like concentrator or front-end processing, users must be able to integrate basic system software functions (task scheduling, for example) into their realtime software while retaining control over methods of error recovery and other operations.

The last of the three application requirements for small computers described in Figure 1—the need for highly reliable equipment combined with the availability of good service even in remote locations—is especially important in communications' applications. Communications' methods which detect all errors are obviously important. In many of these applications if the system is unavailable for any substantial period of time, the user is inconvenienced and important applications may be jeopardized. Consequently, to insure the acceptability and integrity of these systems, both the supplier and the user must provide adequate maintenance and support of the small computer installation.

The Data Acquisition and Control Application

Data acquisition is a critical task in many applications. The general purpose, small computer is an excellent system to perform this task because of its ability to respond quickly to external events, its programmability, and its computational capability. These applications include:

- Process control
- Laboratory automations

- Discrete manufacturing control
- Data collection and online inspection and testing
- Control of materials handling systems, and others

The Data Base

In this application, data is usually gathered from sensors and instruments which are both unique to the process and process variable. The computers used must interface to a variety of instruments and special-purpose devices, and the software operating system supplied must support these devices. Often, the user gathers and enters data manually from special purpose terminals; this is a common practice in some discrete, manufacturing quality control and machine-monitoring applications.

The following characteristics of the data base are common to data acquisition and control applications:

- It is maintained in realtime
- It is shared and operated upon by a variety of programs
- It must often be maintained for long periods of time
- It must be protected from inadvertent destruction
- It is subject to on-demand display, interrogation, and—sometimes—change by operators

Response Time

For data acquisition and control applications, response time to events is an important design consideration in hardware, software, and overall system structure. Data must be in the right place at the right time. Using dedicated small computers at the data source is one way to insure that most of the response requirements are fulfilled; sometimes, the application needs data from remote computers for a quick decision. For example: rescheduling an application when a failure is detected may be time-critical. A key point in the structuring of the system is how quickly central data bases or remote data bases can be accessed.

External events may trigger or time some data acquisition programs. In these cases, the application requires a realtime

operating system with rapid scheduling of certain programs.

If the application schedules programs on the basis of complex events rather than clock time, the system—with minimal delay—must be able to detect such events, notify waiting programs, and activate overall system responses.

Concurrently, other applications are not time-critical, and the system must serve them without introducing the overhead in software or hardware that is required to support the time-critical applications. Flexibility and easy system application tailoring are essential characteristics of the combined time-critical/non-time-critical environment. Data acquisition and control—whether in a small OEM instrumentation system with an imbedded small computer, or in a large process control system with sensors and operator communications' devices scattered around a plant—poses a particularly demanding set of requirements for a general purpose, small computer system.

Hardware and Software Structure

Figure 7 shows a block diagram of a single computer data acquisition system. To control precisely the time between sensor scans, an external signal may trigger the data acquisition cycle. Such precision control over realtime programs is a hardware requirement of those small computers which are suitable for this application.

The data acquisition block is a set of programs which carry out the following:

- Scanning sensors
- Converting input values to proper engineering units
- Limit checking to detect alarm conditions
- Designating a set of alarm programs to respond—within specified limits—to these violations
- Data smoothing
- Controlling output to the instrument or process
- Detecting more complex events

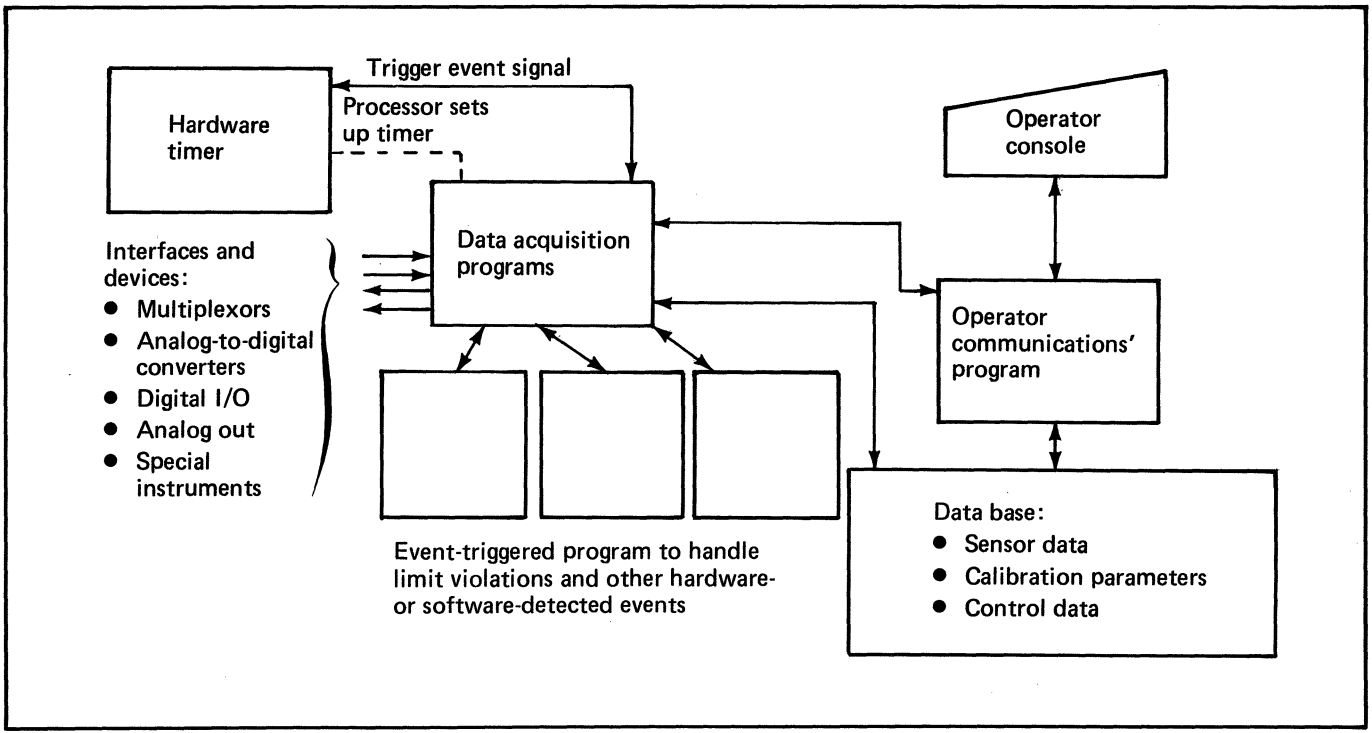


Figure 7. The data acquisition and control application (1 of 2)

Data acquisition and control involves sensor-based input and output devices—including special instruments—together with very time-critical programs. Often, operator communication with the data base (for inquiry, control over tasks, changing of parameters, and other functions) runs concurrently with the data acquisition and control.

The result is a critical set of software and hardware needs in an environment which often demands almost complete availability of the system.

Figure 7. The data acquisition and control application (2 of 2)

Operator interaction with this online system is a common requirement. This interaction involves:

- Inquiry into the data base (status of a variable, history of a variable)
- Modification of the data base (taking a point off scan, modifying calibration parameters)
- Invocation of programs that provide reports or useful computations to the operator

The many programs intermix in time in the sequence of task executions illustrated in Figure 8.

Where the figure shows methods executing asynchronously, the small computer operating system is multiprogramming them. That is, only one program at a time is actually executing; but when one program is blocked—waiting for input data or for another program to supply it data—the system runs another waiting program. Such a multiprogramming operating system simulates parallel program execution.

Software Support

The demands on the software are exacting because programs are communicating extensively with one another (e.g., limit checking occurs only after a point has been scanned; this program in turn notifies alarm programs if the data scanned exceeds a limit). The control program executes less often than the data acquisition programs and, furthermore, operates only on certain data points. The supply of points, for which the system should carry out feedback control, varies from time to time; the data acquisition and smoothing programs communicate this information to the control program. In effect, each program is receiving data from other programs in an unpredictable sequence; consequently, each program operates asynchronously. The arrival of data from one program is the event that triggers the execution of another.

This sequence of transactions imposes the following requirement on operating system software:

> The small computer operating system must support a set of rapidly-responsive, tightly-coupled programs which can share data, govern access to data, and control one another's scheduling.

The set of tightly-coupled programs that carry out this application is termed a task set. In addition to sharing data, the tasks control one another in the following manner: in the process of sharing data from the data base, tasks must coordinate with each other so that one task does not change the same data that another task is in the process of reading. The software operating system must provide the controls to enable the user to govern these functions efficiently and effectively.

When the small computer is part of a smaller, dedicated instrumentation application, the software demands are less elaborate. Nonetheless, the critical requirements of fast response, task communications, data sharing, and other functions are still present. To be competitive, the application and system software must reside in a relatively small main storage. Economics, then, dictates an additional software requirement:

> In some dedicated small realtime applications, the operating system's essential components must be available for imbedding in the dedicated application.

Indirectly, this requirement implies that a small, efficient operating system will result when the essential features of task switching, interrupt response, and intertask communications are built into the system hardware.

Unlike the communications' applications, data acquisition and control applications often use extensive application software. In the latter case, the critical requirements must be imposed on a comparatively general-purpose operating system.

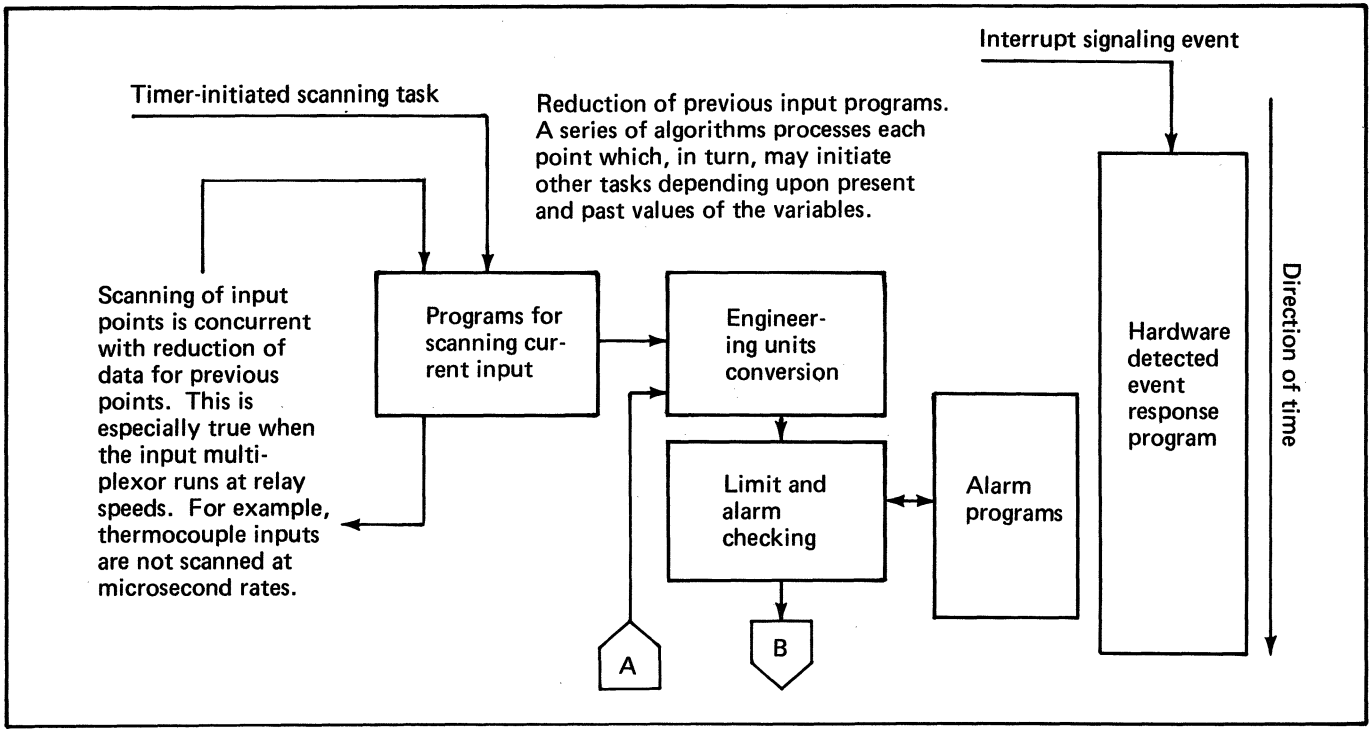


Figure 8. Set of concurrent tasks to carry out data acquisition and control (1 of 2)

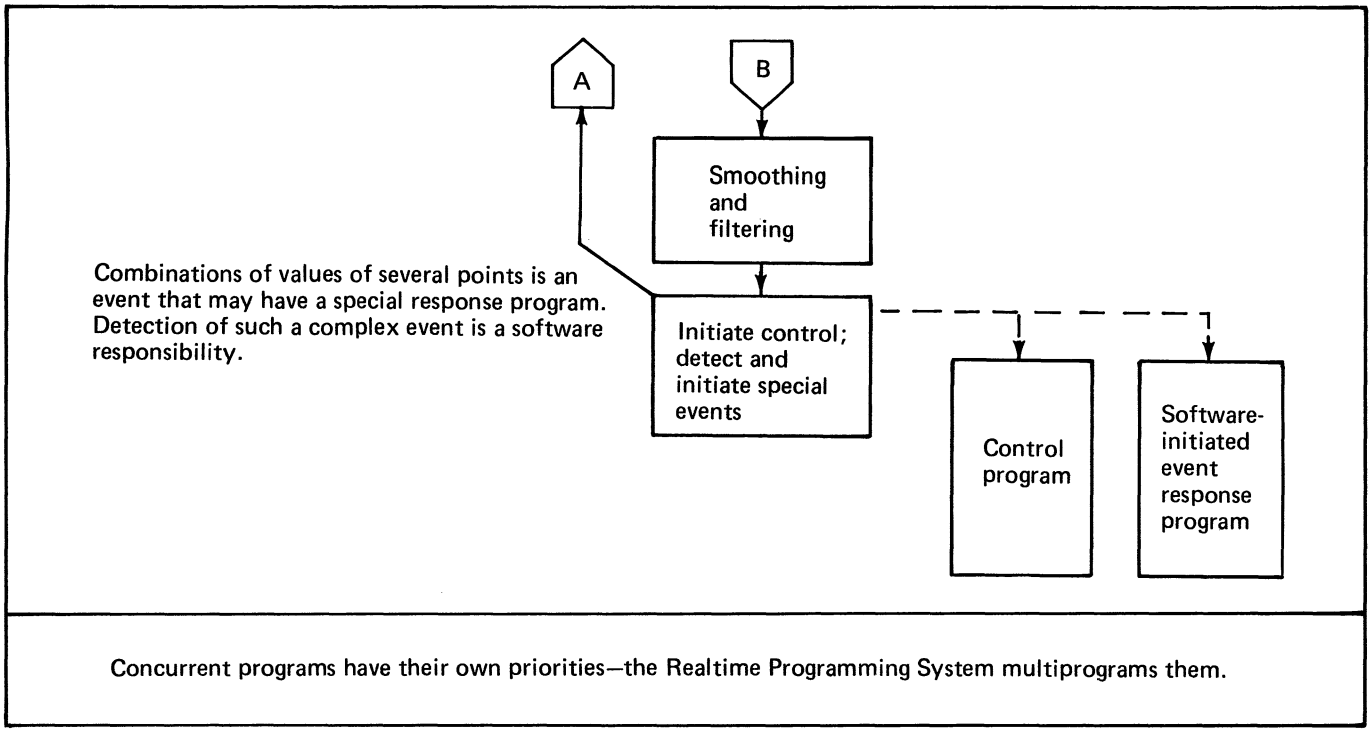


Figure 8. Set of concurrent tasks to carry out data acquisition and control (2 of 2)

Summary of Application Needs

Small computer applications vary so widely that it is not practical to list all of their characteristics and needs. In the future, the number and types of these applications will expand further. This chapter has identified some of these needs and characteristics by examining, in detail, four discrete applications. A summary of these application needs is listed in Tables 1, 2, and 3.

Although these needs are listed separately, users must integrate them carefully to realize the objectives discussed earlier in this chapter. For example, users who integrate complex, realtime software modules into the operating system must be in a position to maintain that software themselves; consequently, in some instances, they must have operating system source code available. Conversely, the smaller user may have vendor-provided maintenance but still require access to the software to make modifications resulting from the addition of OEM device routines. This access should be on a continuing basis to prevent conflicts or difficulties when versions change or updates are added to the code. Most importantly, users must know that the system software is so designed and structured that new operating systems will build on previous ones and—if they want to take advantage of new features—will not require them to redesign their application software.

In contemplating installation of a small computer, users must evaluate the system's ability to meet all of their needs in the context of their application, however that application might be defined:

- A single standalone small computer system
- A long-term product line with small computers as only one component
- A company- or plant-wide application involving many, communicating small computers

The IBM Series/1 small computers are general purpose systems designed expressly to meet the needs of a broad spectrum of applications. The next chapter examines the architecture of the Series/1 and shows how each feature responds to a variety of application needs.

Hardware Characteristics

- A small computer family whose architecture is economically compatible with large and small systems
- An extensive instruction set to support bit and byte manipulations and a variety of data formats
- A storage organization which facilitates multiple tasks
- Good support for switching between multiple tasks with a minimum of overhead
- Fast and efficient response to external interrupts
- An architecture which can support large, main storage devices
- Storage protection
- Efficient and general, programmed and direct, main storage access input/output
- Extensive and flexible communications' interfaces
- Convenient OEM interfaces which preserve system self-diagnosis
- A variety of data processing peripherals; compatibility with special-purpose OEM peripherals
- A variety of sensor-based input/output devices; compatibility with special-purpose and OEM devices

Table 1. Integrated system of hardware needs

Software Characteristics

- System software support for a set of tasks which cooperate to carry out the application
- System software support for a set of tasks which have widely varying response times
- Operating system support which allows efficient realization of task sets in either small or large systems
- System software support for creating, accessing, updating, and protecting data files
- Good control over system resources needed by tasks—including resolution of conflicts among tasks competing for the resources
- System software which permits addition of user-written modules for support of special devices
- Realtime support for scheduling tasks on the basis of time, internal events, or external events
- Effective support for data communications and control among tasks
- Support for application software written in assembly language where efficiency of code or speed of execution is a paramount consideration
- Support for application software written in higher-level languages which support structured programming
- Background computational capability, especially for program preparation
- Ability to mix assembler language routines with higher-level languages to save storage space and reduce programming time

Table 2. Integrated system of software needs

Maintenance and Support Characteristics

- Modern hardware design and packaging for inherent reliability and availability
- Readily available maintenance, including remote locations and other countries
- Variety of maintenance plans to suit the widely different applications of small computer systems
- Extensive self-diagnosis to minimize repair time
- Extensive self-checking to permit detection and localization of hardware problems, especially when multiple device types or attachments are present or the system is linked to other systems
- Software maintenance compatible with the needs of both small- and large-sized users of small computers
- Access to highly-trained software and hardware engineers for support of critical applications
- Explicit long-term maintenance commitment for available operating systems and hardware

Table 3. Integrated system of maintenance and support needs

2 Overview of the IBM Series/1

IBM designed the Series/1 specifically to meet the integrated application needs of small computers—hardware, software, and maintenance support. This chapter presents an overview of the IBM Series/1 and demonstrates how that computer satisfies these requirements.

Series/1 Architecture

Because the IBM Series/1 small computer is a *system* and not simply a set of independent *products*, it is important to recognize two things:

1. The architectural features of the system
2. The specific features of those individual components that are available

System Architecture

The term "architecture" refers to the overall organization of the Series/1. It insures that:

- Individual products in the system integrate closely
- Software systems integrate efficiently with hardware

- Users with different software needs can customize the system and still use the hardware efficiently
- Users with unique hardware devices can integrate them into the system efficiently without losing the advantages of self-diagnosis and other features of the system
- Future changes in technology may be more easily incorporated into the system without obsoleting existing system designs

In other words, a good architecture for the computer hardware, software, maintenance, and support insures that the user will experience a minimum of "future shock".

The overall Series/1 shown in Figure 9 consists of:

- A family of processors and input/output devices integrated with a family of software support
- Hardware and software error checking and self-diagnosis
- Backup by IBM maintenance and support personnel in many countries around the world

The Processors

The family of processors is microprogrammed; this permits the user to employ a rich instruction set of over 200 individual instructions facilitating application development and minimizing program size.

One processor in the Series/1 family, shown in Figure 10, indicates some of the important characteristics of the entire family. The processors are rack-mountable and provide slots or positions into which the user can plug printed circuit cards. Each card implements storage, input/output interfaces, or other options. As indicated, the user can extend the number of slots in a straightforward manner, thereby economically changing the size of the system. Maintenance—by exchanging the printed circuit cards—is simple; and, when coupled with the extensive self-diagnostic capability of modules in the Series/1, requires a minimum of time.

Input/Output

The input/output system provides a fast channel (which supports both direct access storage devices and direct

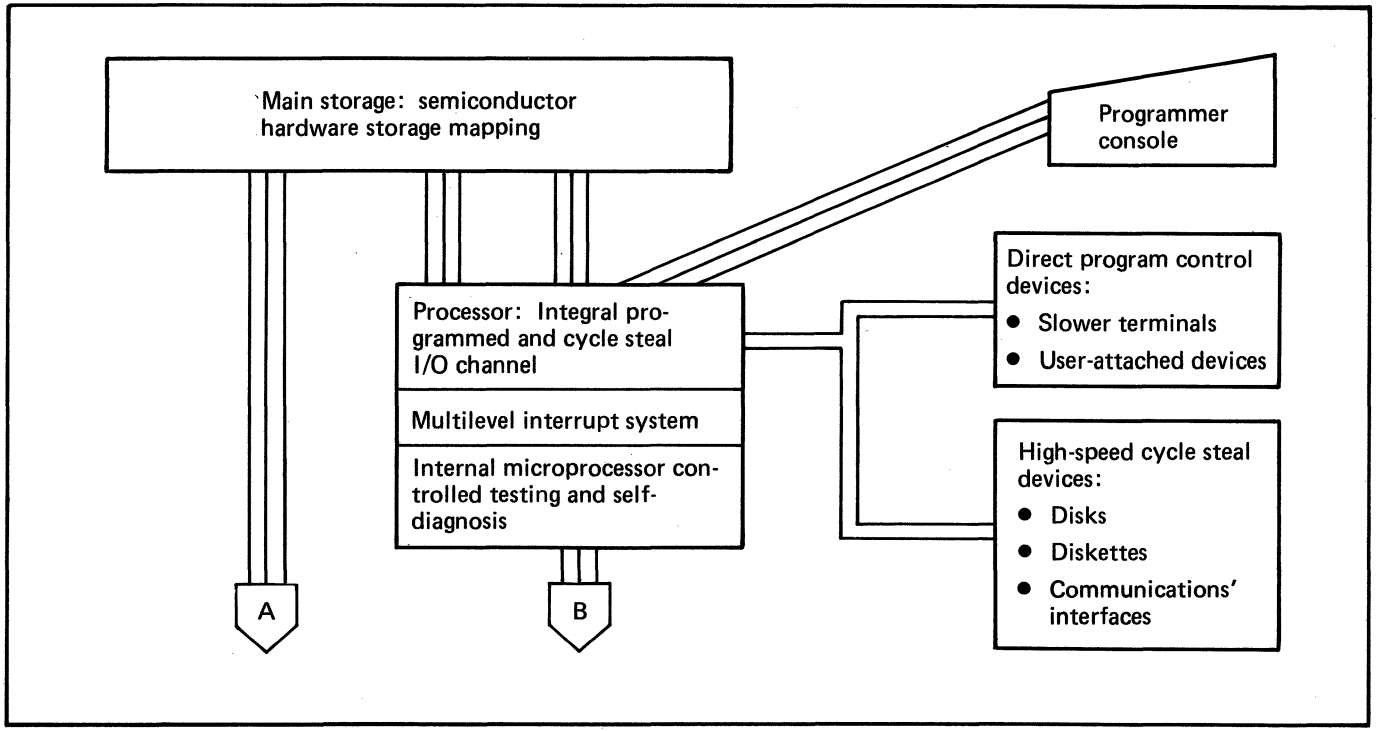


Figure 9. IBM Series/1: an integrated system of hardware, software and maintenance elements (1 of 3)

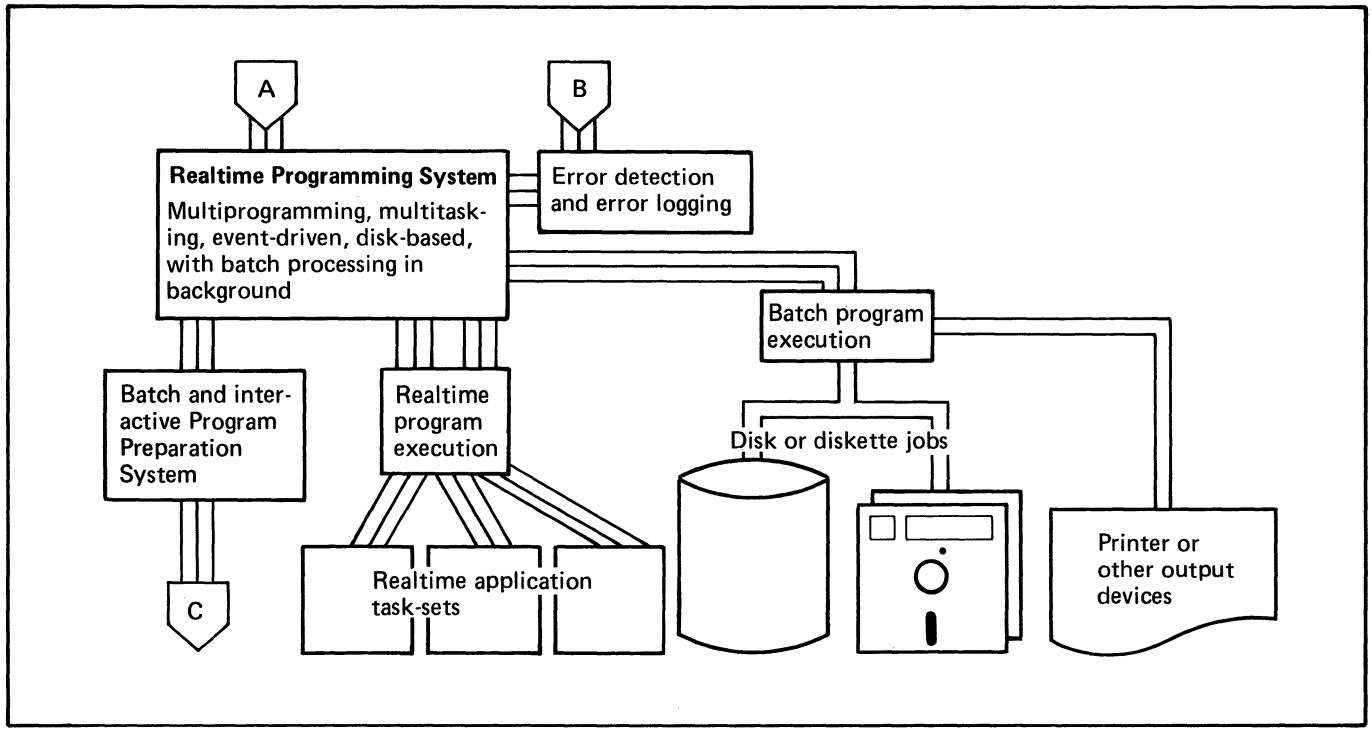


Figure 9. IBM Series/1: an integrated system of hardware, software and maintenance elements (2 of 3)

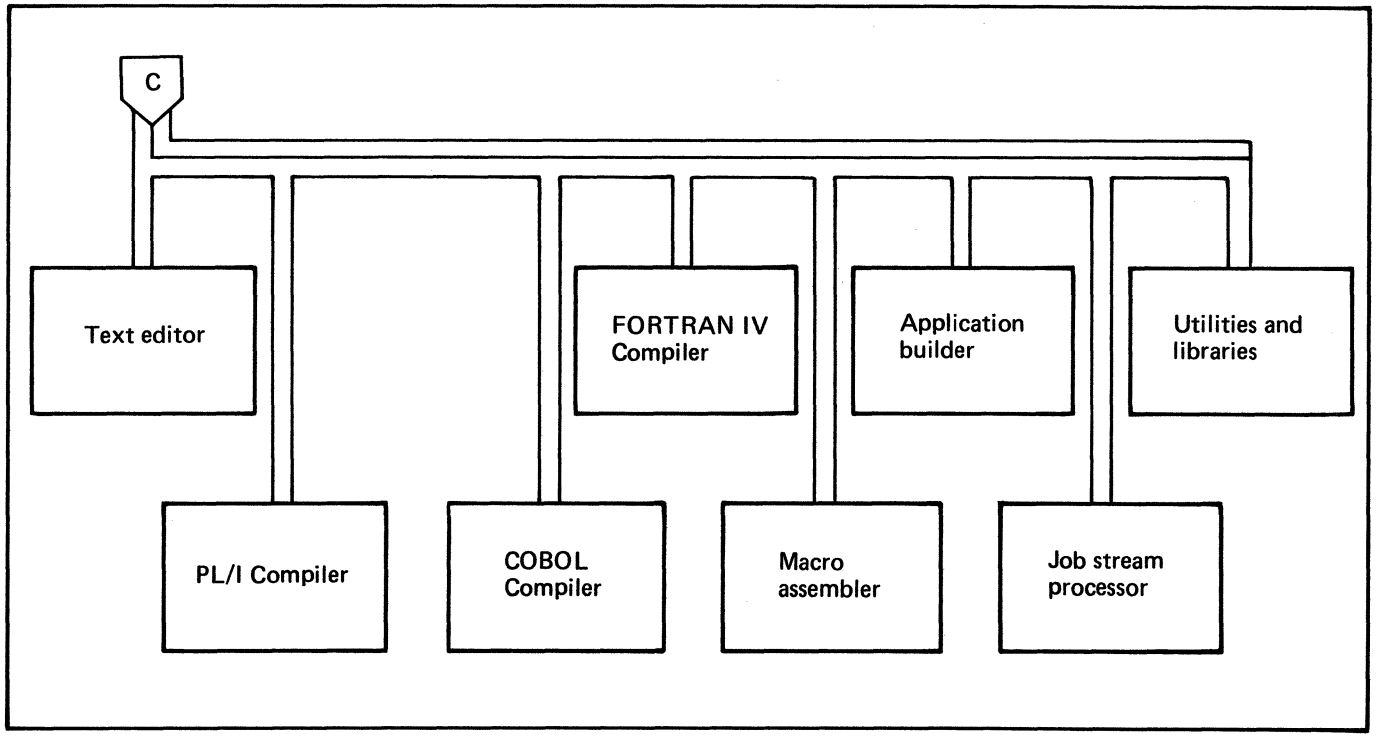


Figure 9. IBM Series/1: an integrated system of hardware, software and maintenance elements (3 of 3)

program control devices) into the system. Input and output can be:

- A single item at a time under program control or interrupt control
- Multi-item transfers in parallel with program execution on a cycle steal basis, or
- Very fast, high-volume transfers on a "burst mode" basis at the maximum storage access rate

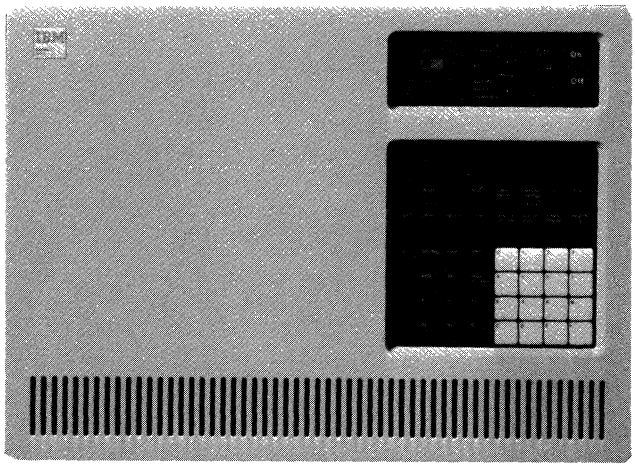
By duplicating appropriate hardware registers on each level, the multilevel interrupt system achieves very fast response to external events.

Main Storage

Semiconductors are used exclusively for the main storage of the Series/1. Flexibility in addressing main storage is achieved through a variety of addressing modes in the instructions. Recognizing the fundamental characteristic of small computer applications—"a set of cooperating tasks"—the IBM Series/1 designers deliberately chose an organization of storage which efficiently supports the needs of individual tasks. At the same time, this organization permits multiple tasks to coexist in storage and insures that realtime operating systems support them efficiently.

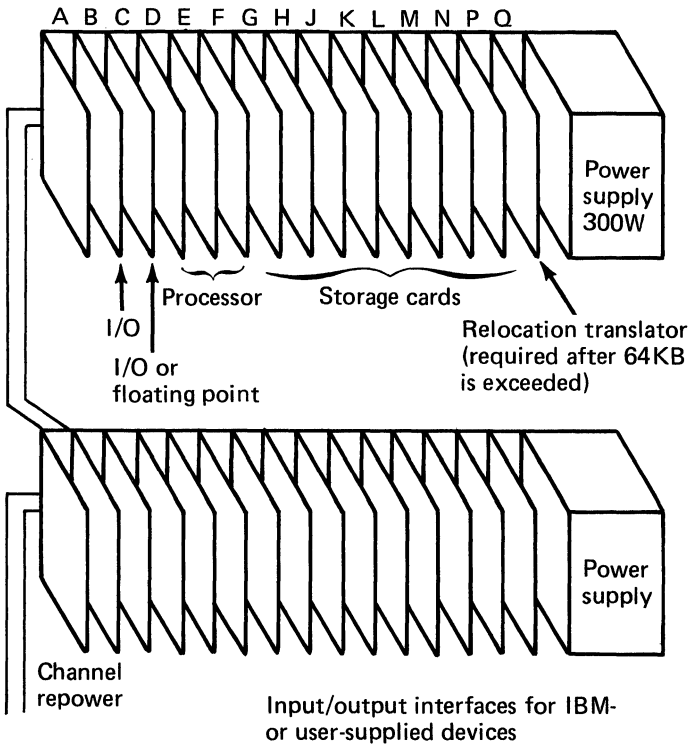
Address Translation

Individual tasks generate addresses within the task itself; special address relocation hardware then maps the tasks into actual main storage hardware addresses. Figure 11 shows this process. User programs have a 16-bit address space available to them, and may reference all data and locations within the program with an address relative to the start of the program. Since several programs reside simultaneously in physical storage, typically, an individual program does not reside at the start of the physical storage. It is the function of the relocation hardware to translate each address generated within the program at execution time into the actual physical storage location of the referenced item.



- 19-inch rack mountable
- TTL bipolar LSI logic
- Over 200 instructions on larger processors
- Programmer's console—optional
- 660 nanoseconds or 800 nanoseconds cycle time, depending upon the model of the processor
- 16-bit word length, one parity bit per byte on data bus; no parity on address bus
- Storage protection against writing or access on some models of the processor
- Main storage bit, byte, and word addressable in appropriate instructions
- Maximum storage size from 64K bytes to 256K bytes depending upon the processor model

Figure 10. Features of the IBM Series/1 processor family (1 of 2)



- Storage mapping via relocation translator card to map multiple user tasks into physical memory space (on some models of the processor)
- Power supply:
  - Battery backup optional
  - Power-fail/thermal warning detection
- Input/output expansion units for additional devices:
  - Separate power supply
  - Channel signal repower unit
- All options can be added in the field with pluggable cards

Figure 10. Features of the IBM Series/1 processor family (2 of 2)

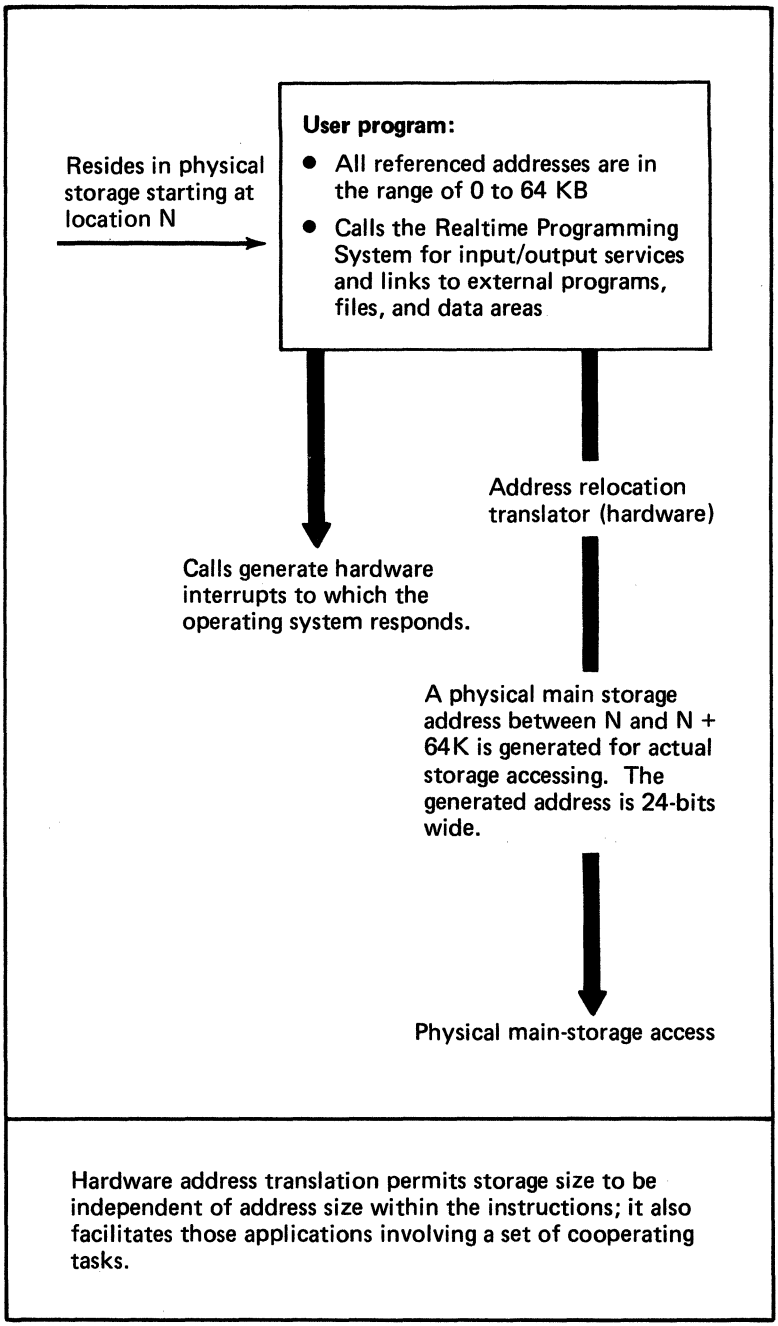


Figure 11. Address relocation for user programs

For communications with other programs and input/output devices, special instructions (called Supervisor Call instructions) generate hardware interrupts. The operating system or appropriate control program module responds to these interrupts.

Without using relocation hardware, users can create economic small systems whose task addresses are the same as their physical storage addresses. Larger systems can take advantage of relocation hardware to combine a number of tasks cooperating to carry out the application.

Address translation provides the basic *hardware* method by which small and large systems can be included within the same overall hardware, software, and maintenance organization or architecture. *Software* systems (operating systems) for smaller systems will not support address translation for two reasons: 1) it is not needed functionally for such systems, and 2) the size and cost of the software make address translation uneconomic for these applications. The architecture of the software itself, however, is consistent with address translation. Therefore, when creating special operating system software, users can take advantage of the hardware architecture, permitting the development of larger application software systems.

The Series/1 16-bit word length conforms to the general characteristics of the application. This word length contributes to efficient instruction and data storage, and provides a convenient address size for application tasks. Because of the relocation hardware function discussed earlier, the 16-bit word length does not limit physical storage size. In fact, the relocation hardware generates a 24-bit physical storage address from each 16-bit address used in an application task. Removing the relationship between physical storage size and the size of instructions and addresses in application tasks insures that users can economically scale the size of their Series/1 processors and applications.

Software Organization

The software needs and characteristics listed in Table 2 of Chapter 1 are, essentially, various aspects of programming

support of hardware. Specifically, in this and subsequent chapters, software discussions will focus on the Realtime Programming System—the primary operating system for Series/1—and its supporting program preparation and language facilities. Figure 9 indicates this overall Series/1 software organization. This organization comprises a general purpose operating system (the Realtime Programming System) which supports:

1. Realtime task sets of the type needed by applications
2. A batch processing or background mode of operation for application software preparation

The Realtime Programming System is carefully integrated with the hardware architecture of the Series/1. In its task usage, the Realtime Programming System takes full advantage of the hardware storage mapping and addressing facilities. It cooperates with self-diagnosis by detecting and logging errors. Its support of a variety of input/output devices is useful both for applications and for user generation of software; in the latter case, batch processing or interactive terminals are utilized.

Control Program Support

Small dedicated applications may not need the full capability of the Realtime Programming System. For these applications, IBM offers a set of modules to provide task management, data processing input/output support, and initial program loading for both disks and diskettes. Users can combine this set of modules, called Control Program Support, with their own application programs to furnish facilities similar to the Realtime Programming System but in a form tailored to each user's dedicated application. Thus, the Control Program Support modules provide the same type of family-size scaling to software that the storage management and addressing architecture provides to the hardware.

Event Driven Executive

A third operating system option exists with the Series/1. The Event Driven Executive in the Series/1 can apply to a

broad range of applications such as data entry, remote job entry, distributed processing, and other commercial applications, as well as typical sensor-based functions like data acquisition, material and component testing, machine and process control, and shop floor control. The Event Driven Executive offers low-entry multiprogramming in a diskette-based system.

Indicative of the integrated nature of the software is the design of the software preparation capability. This function, running under the Realtime Programming System, produces object modules which, in turn, run with the Control Program Support modules in dedicated applications. The user can then prepare software on a large computer and execute it on a smaller, dedicated processor. Similarly, Series/1 offers the user the Basic Program Preparation Facilities which are a set of standalone programs that provide software preparation capability on a machine without the Realtime Programming System.

Higher-Level Languages

The Series/1 gives full support to assembly language programming and several higher-level languages. The IBM Series/1 is unique in the small computer marketplace: by offering a full PL/I compiler, a small computer system has—for the first time—all the capabilities of a modern, fully-structured programming language. FORTRAN and COBOL higher-level languages are also available. This range of programming languages allows users to select the language most appropriate for implementation of their application tasks.

Self-Diagnosis

The third element of the small computer design (as discussed in Chapter 1) is maintenance and support. The IBM Series/1 processor is microprogrammed. The interface to almost every module in the system contains a microprocessor—a stored processor implemented with large-scale, integrated technology in a single, small package. These

microprocessors provide both the logical control and coordination between the device and the processor as well as a diagnostic capability for the system itself (Figure 12).

IBM designed the Series/1 architecture to enable each module to logically disconnect from the system for diagnostic purposes. The microprocessor associated with each feature exercises the functions of that feature to diagnose any problem to the level of that module. If modules pass this self-diagnostic test, they then interconnect and the microprocessors carry out a second level of diagnostics. For example, the system performs communications back and forth between appropriate modules to diagnose any problems associated with the interaction of those modules—again to the module level.

If module interconnections pass this second-level, self-diagnostic test, the system interconnects and initial program load (IPL) occurs. Self-diagnosis then passes from the hardware level to the software level. Here, modules within the Realtime Programming System constantly monitor operations, detect and retry errors, and—optionally—log all detected errors to disk or diskette for later examination by maintenance personnel.

Self-diagnosis at the module and system level minimizes the difficulty in detecting and isolating the source of difficulties and, in turn, minimizes the mean-time-to-repair them. The functional and economic importance of this module by module self-diagnosis cannot be overemphasized.

Maintenance

IBM customer engineer and maintenance support backs up the self-diagnostic capabilities of Series/1. User maintenance and support needs vary considerably. For those users producing products or systems involving the IBM Series/1 who wish to do their own maintenance of hardware and software, IBM provides complete hardware and software documentation together with detailed training courses comparable to those supplied IBM customer engineers. Hardware diagnostic terminals and special diagnostic software are an integral part of the Series/1.

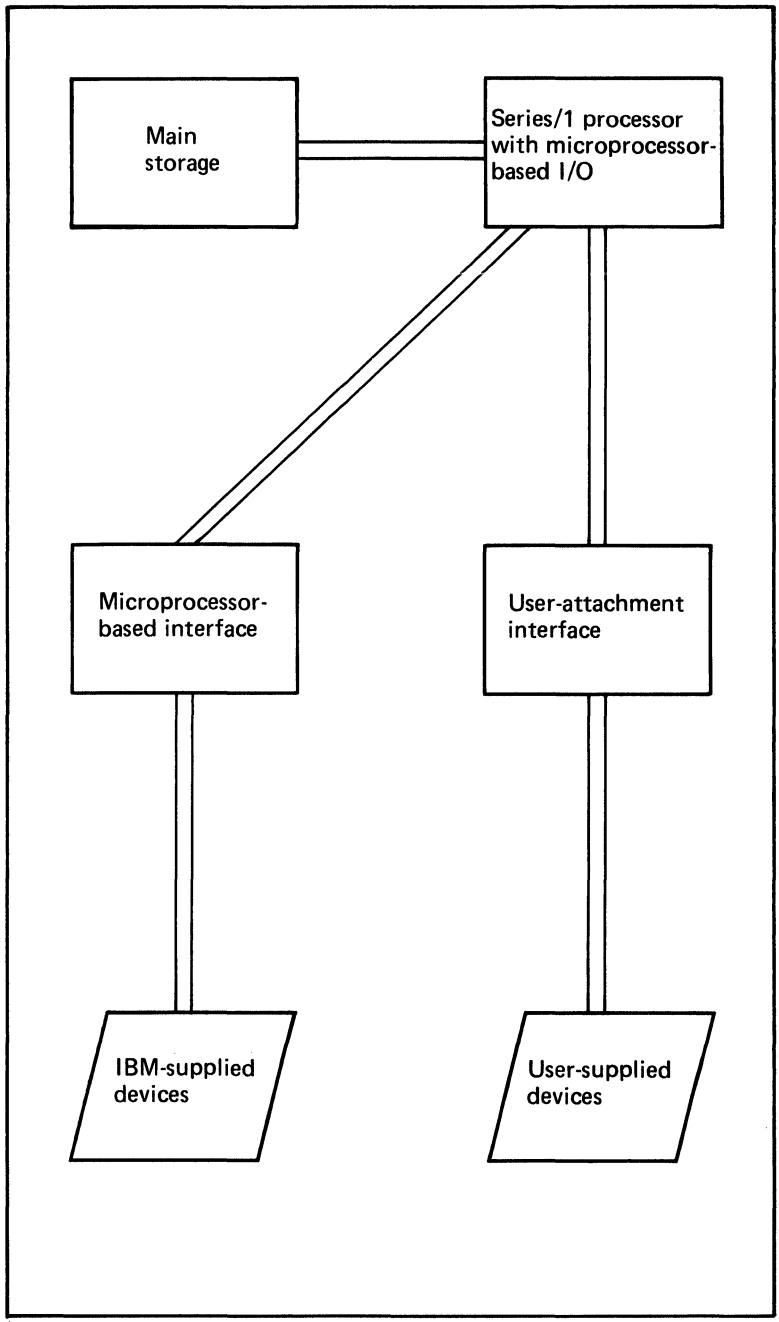


Figure 12. Series/1 self-diagnosis (1 of 2)

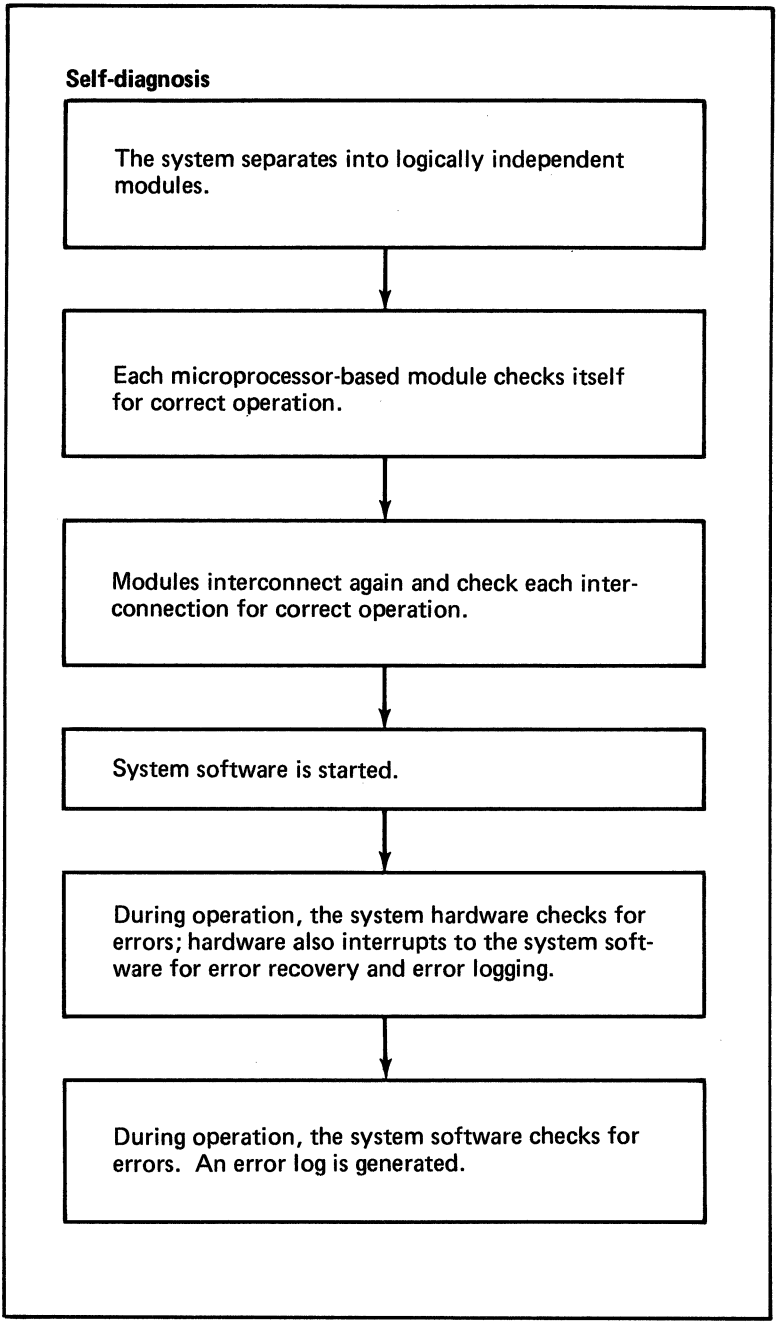


Figure 12. Series/1 self-diagnosis (2 of 2)

Software maintenance is always a critical item. IBM-provided Series/1 software is so structured in its design that user documentation is extensive, comprehensive, and easily understandable. IBM Series/1 program logic manuals provide:

- Detailed descriptions of each software module method-of-operation, including HIPO charts (a hierarchical structure of charts that shows the functions of the various programs' components including their input and output functions)
- Descriptions of each data area
- Diagnostic aids
- Debugging hints
- Detailed logic of the module

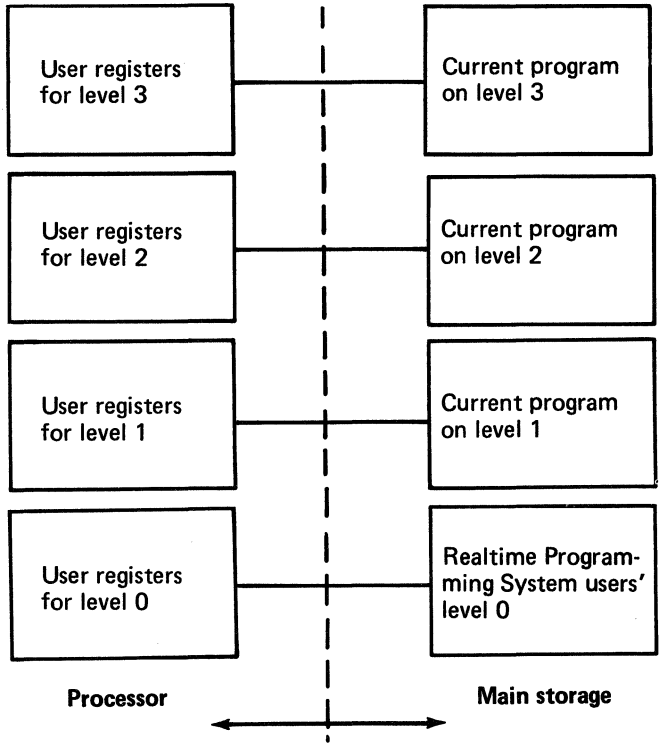
If Series/1 users want to do their own maintenance, IBM will furnish source code for most IBM-supplied software.

However, most users do not, themselves, maintain either hardware or system software; they need quality maintenance by competent personnel on a world-wide basis. As an integral part of the overall Series/1 product-line support offering, IBM makes available hardware maintenance, software maintenance, and special customer engineer support for the Series/1.

Hardware and Software Support of Multiple, Cooperating Application Tasks

The Series/1 processors provide hardware support for multiple tasks through the system's relocation hardware. Efficient utilization of multiple, cooperating tasks also requires that the system be able to multiprogram those tasks. When there are several programs in main storage, each assigned to one hardware priority level, the processor provides hardware support for task switching (Figure 13). For each level, the system provides a separate set of hardware registers including:

- Eight general purpose registers
- Floating-point registers



Each hardware priority level has its own set of user registers:

- Eight general purpose registers for data, addresses, indexing, displacements, and other information
- An instruction address register or program counter for each level
- A status register for flags and error reporting
- A storage address key register for storage protection

Switching from a program on one level to a program on another level does not require saving or restoring the duplicated user-registers. Hence, multiprogramming of tasks on different priority levels is rapid.

Figure 13. Support for multiprogramming of multiple user tasks

- Arithmetic and status indicators
- An instruction counter
- Information for storage protection

Switching from a task on one hardware level to a task on another level does not require saving or restoring these register contents; consequently, the switching is very rapid.

Interrupt System

Normally, the system connects input/output routines to hardware priority levels which are different from those of application programs. As program execution switches back and forth between the application priority level and the input/output level, this structure permits interrupt-driven input/output operations to occur—with a minimum of overhead—concurrently with application tasks. To facilitate fast response for critical tasks, application tasks themselves can reside on different hardware levels.

The combination of hardware memory management and user registers duplicated on each hardware priority level allows a fast response to realtime events through cooperating tasks. A user can, of course, configure such a set of tasks with the Control Program Support modules or another customized control program. The Realtime Programming System facilitates such applications using the organization shown in Figure 14.

Multiprogramming and Multitasking

Multiprogramming is the execution of two or more tasks concurrently. The system accomplishes multiprogramming by switching the processor use to the higher priority of two waiting tasks or—when a higher priority task cannot continue execution, e.g., because it is waiting for input data—to a lower priority task. The Series/1 Realtime Programming System performs multiprogramming among all tasks in main storage. In addition, it permits multitasking; this process consists of generating several secondary tasks from a single main task.

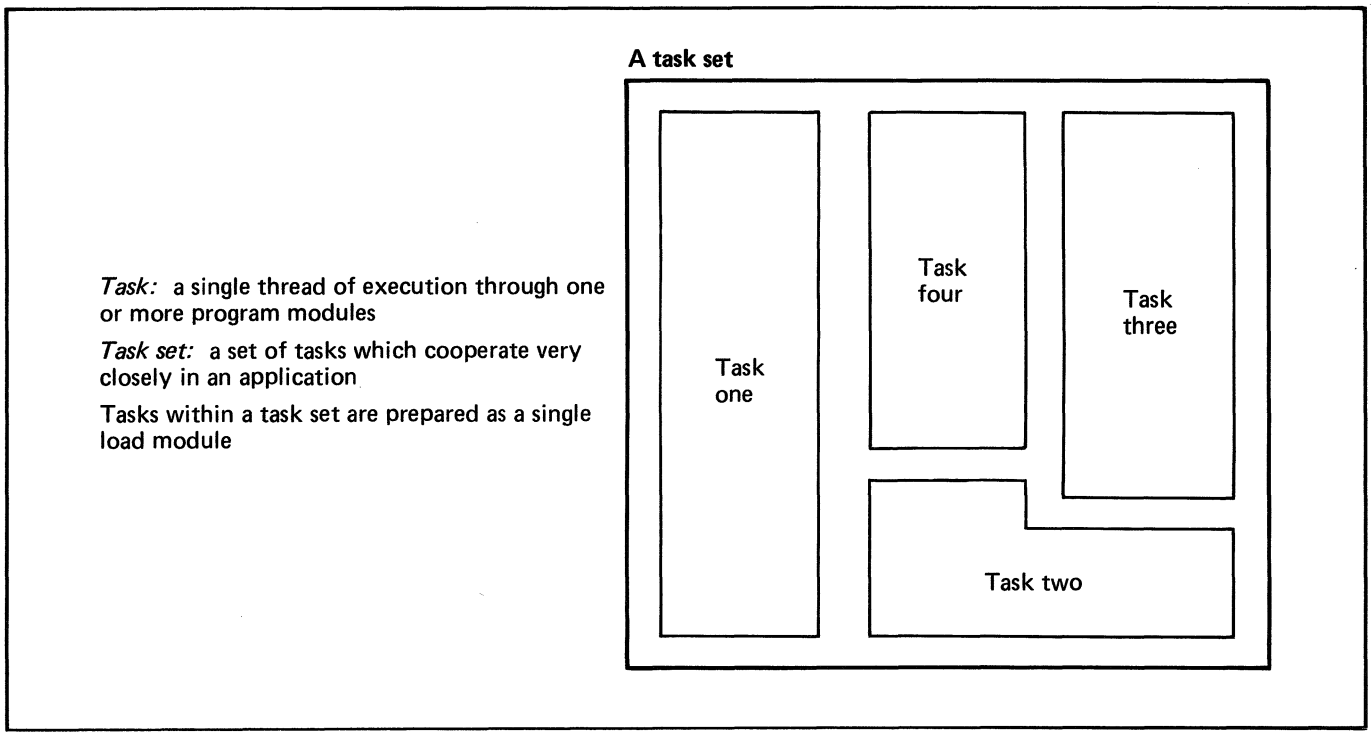
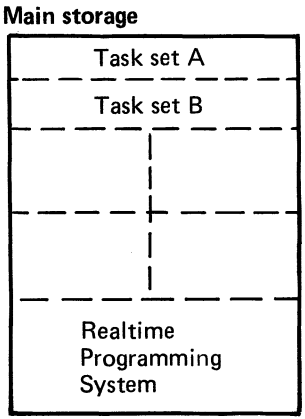


Figure 14. Task sets and the organization of main storage under the Realtime Programming System (1 of 2)

Each task in a task set uses the Realtime Programming System services to:

- Schedule other tasks
- Perform input or output operations
- Communicate with other tasks
- Call routines in other tasks



One task set at a time resides in each partition. Requests for execution of other task sets in this partition are queued by the Realtime Programming System.

A task or task set may reside on any hardware or software priority level.

The Realtime Programming System is a multiprogramming, multitasking, event-driven, disk-based system. It manages all physical resources.

The Realtime Programming System multiprograms among task sets in the partitions according to their priority.

Figure 14. Task sets and the organization of main storage under the Realtime Programming System (2 of 2)

The central concept of this system is the "task set" which is a collection of closely cooperating individual tasks, programs, and data. At program preparation time, the tasks within a given task set link together into a single unit for loading from disk. Once the task set begins execution, the operating system schedules and multiprograms individual tasks.

Storage Management

To give the system designer good control over response time of application tasks, the system uses a fixed partition organization of storage as shown in Figure 14. A dynamic partition option is also available. A fixed partition is a contiguous area of storage in which one task set at a time can execute. By specifying the priority of tasks and assigning task sets to appropriate partitions, the system designer can insure that critical tasks will respond when they are needed. The operating system provides all the facilities needed for queueing task sets that are waiting to execute.

If a task set requires a fast response to an event, the system might assign it permanently to a partition so that the task set need not be loaded from disk when the event occurs. If several task sets are time-critical—but not so critical that they must be permanently resident in main storage—the system may assign them to share exclusively a partition. This minimizes conflict when the task sets are ready for execution.

The system may assign several less time-critical task sets, including background program-preparation task sets, to one partition. With this organization, the Realtime Programming System gives the application designer full control over tasks.

When response time is less critical, it is often convenient to allow the system itself to determine the size of the partitions; this procedure enables the partitions to fit the task set scheduled to execute without using unneeded space. The system then has free space that can be used by another task set. Allocating an area of space in which partitions can be created on demand is called dynamic storage management and is supported under the Series/1 Realtime Programming System. The advantage of this convenience is offset by the

additional time required to initiate execution of a task in such a dynamic partition. Because dynamic partitions are not fixed, once the system initiates a task, it cannot be rolled out of main storage for higher priority tasks. Consequently, dynamic storage management is useful only for tasks that are not time-critical.

Many applications, however, combine time-critical and non-time-critical tasks. Because the Realtime Programming System supports both fixed and dynamic partitions, system implementors can choose those combinations of fixed and variable partition space which most efficiently support their applications.

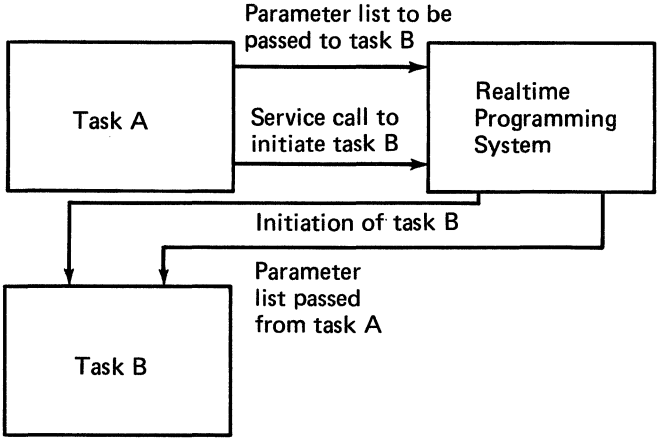
These features are included at system generation time. Users who do not need dynamic partitions do not incur any overhead in the size of their operating systems because the code for the partition function is not added to their generated systems.

Intertask Communications

Tasks within a task set communicate extensively. In fact, the system usually combines multiple tasks into a single, specific task set because they share data and control. In addition, the related tasks interact so often that they can respond within the required time frame only if they are in the same partition. In actual applications, tasks communicate with one another continuously; this fact is the dominant consideration when a user designs software for realtime applications. This communication consists of passing data items, groups of data, event signals, and control back and forth among the various cooperating tasks which are implementing the application.

As illustrated in Figure 15, the Realtime Programming System provides different methods of communications among the tasks. Optional parameter lists facilitate passing control from one task to another. These lists may supply a limited amount of data needed for the task to begin execution (files or global areas transmit more extensive amounts of data).

Parameter lists: are passed to programs which are initiated by a communicating task.



Events: a set of conditions detected by one or more tasks, the occurrence of which can cause execution of other tasks. Interacting tasks can easily synchronize with the event mechanism.

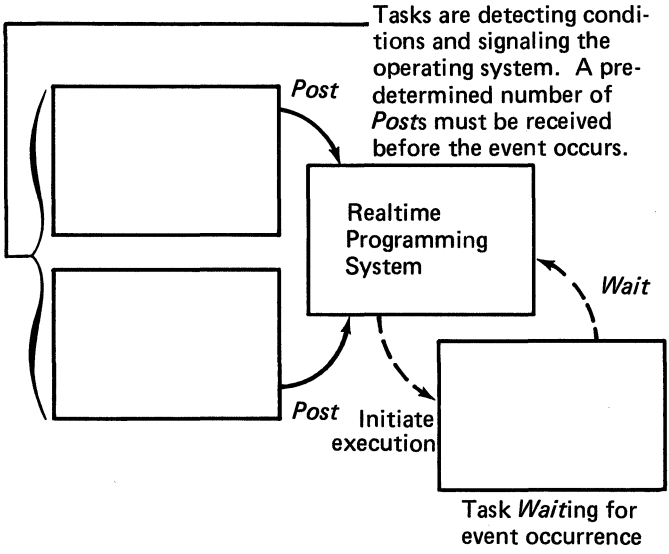
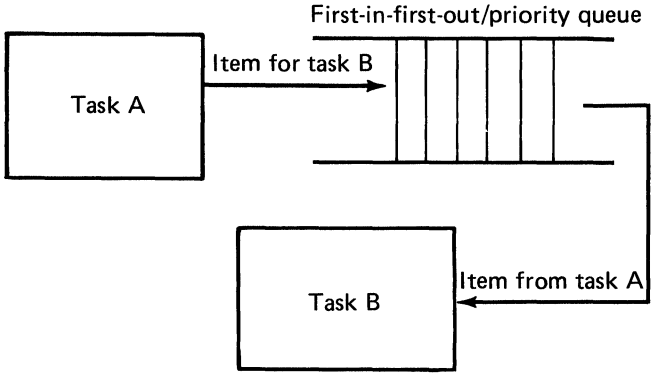


Figure 15. Communications among tasks (1 of 2)

Queues: areas defined in main storage or in auxiliary storage. They contain tasks, enter items into the queues, and remove items from the queues.



Global common area: an area within a partition which may be used by all tasks within a task set.

Each task set may have its own global common area in its own partition.

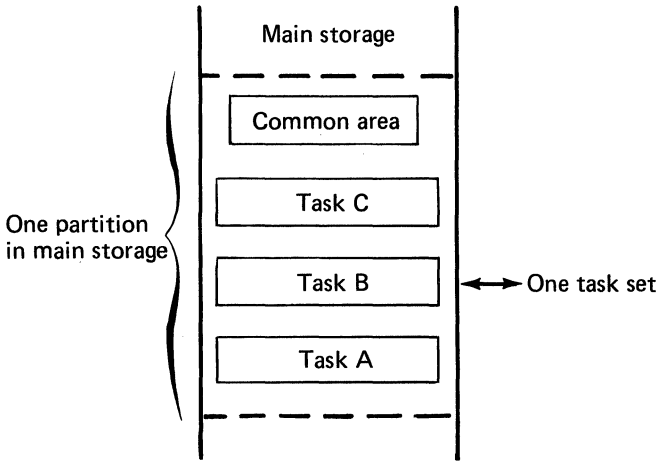


Figure 15. Communications among tasks (2 of 2)

Synchronization among tasks in a task set is possible through:

- Their different priorities
- The scheduling of one another using operating system service calls, and
- Other facilities of the Realtime Programming System: for instance, Wait/Post

Tasks may invoke their own execution when a certain event has occurred once or when it has occurred a specified number of times. Other tasks may detect these events and then inform the operating system (i.e. post the event).

For many applications, like communications, tasks process a number of transactions and pass the results on to other tasks in the task set. Stacks in main storage are a natural communications' mechanism for these actions; both the basic instruction set of the processor and services in the operating system support them.

The operating system also supports sharing of data through an area common to all tasks in a task set—a very useful technique in task management. In fact, the operating system permits task sets themselves to be shared. This means that the programs *within* such a shared task set can be "called" from *other* task sets; the system can then more readily share data areas and common subroutines among a variety of real-time tasks. Communications among tasks is characteristic of many applications but the most appropriate communications' procedure varies from one application to another. Consequently, flexibility of communications in the operating system and control over those communications by the application system designer is:

1. An important attribute of the IBM Series/1 hardware and software
2. A further illustration of the integrated nature of hardware and software

Communications with Remote Devices and Computers

A general purpose small computer must support—in an integrated fashion—hardware and software communications with remote terminals and computers. Figure 16—where three data link structures are described—indicates how necessary the generality of hardware and software architecture is. The Realtime Programming System supports the more standard configurations; for nonstandard configurations, OEM users can integrate the software appropriate to their configuration with the vendor-supplied software. When the application involves significant physical distances, dial-up networks or switched data links are usually necessary. Because of the architecture of the Series/1, customized software should not be necessary for most of these applications.

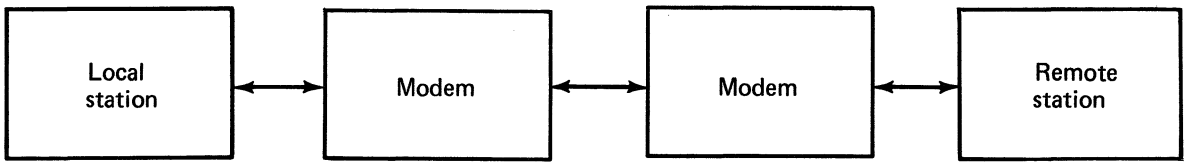
Communications' Protocols

A communications' protocol is the convention by which particular sequences of characters are interpreted and acknowledged. Both the data link structure itself and the protocol or form of communications across these data links vary. Communications' features include:

- Asynchronous communications—single and multiple line interfaces
- Binary synchronous communications—single and multiple line interfaces, plus high-speed single line interface, and IPL capability
- Synchronous data link control communications—single line interface

For asynchronous communications, these features permit both slow- and high-speed terminals to be either directly connected to the Series/1 or connected through modems and switched networks. The two synchronous communications' modes also support Series/1-to-Series/1, and Series/1-to-host communications. The asynchronous communications' attachments provide great program flexibility to the user. Under

A line: is *point-to-point* when a local station is connected to a single remote station. Such a line is *non-switched* when there is a permanent connection between the local station and the remote station through their respective modems, or when the stations are directly connected.



Communications on each type of data link may use different communications' rates and different communications' protocols like:

- The start-stop asynchronous communications' protocol
- The binary synchronous communications' protocol
- The synchronous data link control communications' protocol

Figure 16. Different data link structures (1 of 3)

The primary station: in a *multipoint* data link is physically connected to several secondary stations through their respective modems. The primary station polls the secondary stations using unique station addresses. Only the addressed station can respond to the poll.

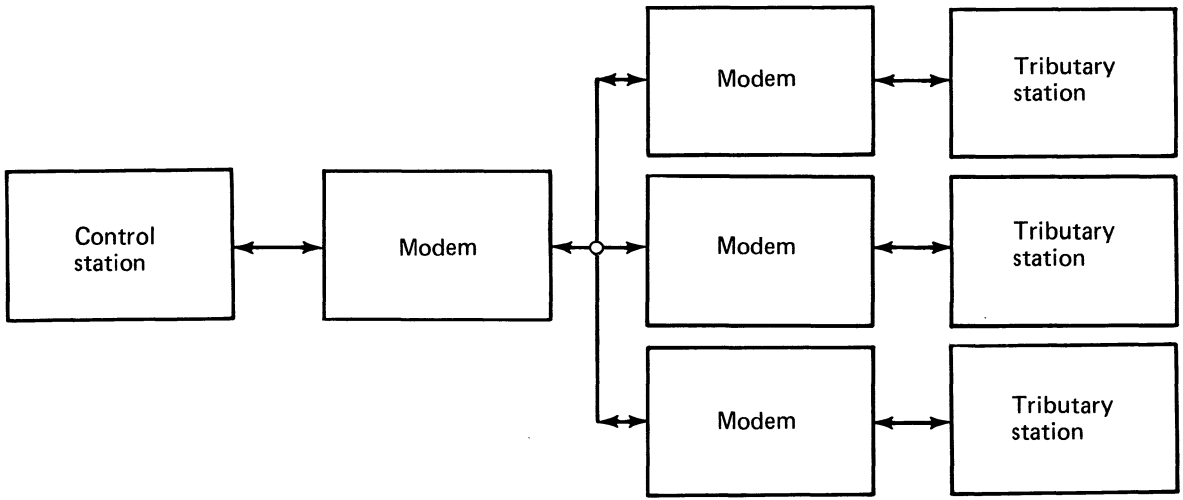


Figure 16. Different data link structures (2 of 3)

A point-to-point line: can be *switched* so that one local station can communicate with *one* of several remote stations after a link has been established between the local station and the remote station. The connection is maintained only for the duration of the communication.

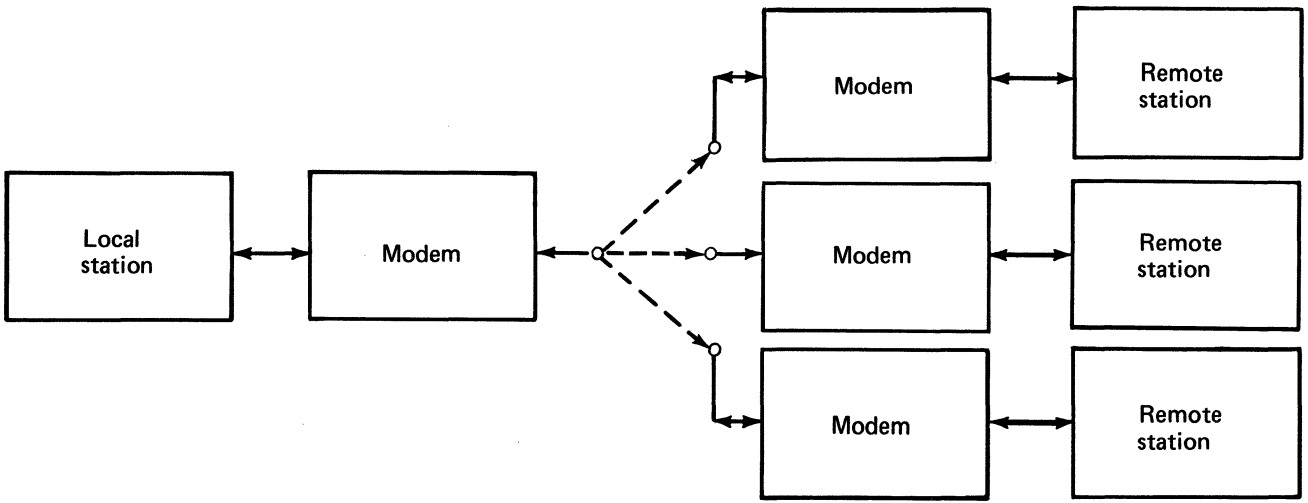


Figure 16. Different data link structures (3 of 3)

program control, the user can select both communications' codes and special control characters to particularize the interface to a specific communications' problem. Under program control, the user can also select communications' rates—up to 9,600 bits per second, and in some cases even 56,000 bits per second—to support the common line speeds.

Similarly, the binary synchronous communications' attachments support both EBCDIC and ASCII codes—again, selected under program control. This modem selects transmission rates by using jumper wires on the interface. The hardware fully supports:

- The binary synchronous communications' protocol including sending and receiving unrestricted binary data (known as data transparency)
- Intermediate block checking of separate error detection information
- Remote loading of initial programs for computer startup or restart. This communications' protocol is most often used in communications between computers, between remote job entry terminals and computers, and similar applications.

The synchronous data link control (SDLC) communications' protocol operates in half-duplex mode, on either switched or dedicated lines, at rates up to 9,600 bits per second. This modern communications' protocol expedites remote communications between intelligent terminals and computers. All the communications' interfaces operate on a cycle steal basis so that minimum processor overhead and interaction are required. Furthermore, software support of communications is enhanced through availability of all cycle steal capabilities including command chaining.

The communications' feature interfaces use imbedded microprocessors for self-diagnostic purposes. The system can easily perform online diagnostic tests of terminals—for example, echoing of test characters.

Communications' Software

Different Series/1 users need different kinds of communications' software support. Some dedicated applications may need custom software. In most cases, however, users require operating system support so they can utilize central error detection and recovery, service calls for input/output, and similar aids. This support is available through the Realtime Programming System, which treats a remote station as a data set—just as it does in other modes of input and output.

Through the operating system, the communications' data sets are opened to establish communications with a given remote terminal or computer. Then, the system accomplishes transmission of data through normal read/write service calls like any other input/output operation. The combination of operating system software and interface hardware carries out the detailed message generation and control functions appropriate to a particular data link. The operating system supports online testing of terminals to insure that communications are proceeding correctly. Because of these hardware and software interactions, the Series/1 fully sustains the communications' requirements of the typical small computer application.

Communications to an IBM System/370

Series/1 can communicate with an IBM System/370 using synchronous data link control or binary synchronous communications. In addition, a special hardware attachment is available to interconnect the Series/1 and the System/370. This interface not only permits the direct coupling of the two systems' channels but also provides the internal logic that enables the Series/1 to recognize the connection as a single device address and the System/370 to recognize the connection as a multiple device address. Software support for this interconnection permits:

1. The down-line loading of programs and data from the System/370 to the Series/1, and
2. The implementation of common distributed system architectures

Auxiliary Storage Devices

Among the more important system needs are auxiliary storage devices suitable for:

- Data storage
- File storage
- Program storage
- File backup
- High-volume, long-term data storage
- Program preparation
- Field modifications
- Other uses

The IBM Series/1 serves these needs with both disk and diskette storage devices.

Disks

The non-removable disks have capacities ranging up to 64 megabytes, depending upon the model. Their interfaces operate on a cycle steal basis and use a thousand-byte buffer which permits instantaneous data transfer rates on the order of a megabyte per second. The interfaces contain a built-in initial program load capability. The disks have moving heads which give a latency time as fast as 9.6 milliseconds, and a back-to-back access time as fast as 27 milliseconds. Where speed of response to auxiliary storage is important, a portion of the disks—on an optional basis—can have fixed heads; this option adds approximately 125,000 additional bytes of storage that these heads can access. Because the moving-head access delay does not affect fixed-head storage, average access time is no more than the average latency. The reader should consult the appropriate reference manuals for the specific maximum storage rates and sizes appropriate to a given configuration.

Diskettes

The second type of auxiliary storage unit available for the Series/1 is a diskette unit which uses a removable one- or two-sided diskette. The diskette unit provides:

- Approximately a one-half megabyte of storage (0.6 megabyte if a 512-byte sector format is used)
- Track-to-track access time of 40 milliseconds
- A data transfer rate of 31,250 bytes per second through the cycle steal interface

The interface supports multiple sector transfers and initial program load capability. As with other devices, the microprocessor controls the diskette with extensive microdiagnostics and constantly-operating error checking.

The high-speed disk is essential to rapidly processed applications; the Realtime Programming System uses the disk extensively.

On the other hand, the diskette unit represents a very useful device for:

- Distributing software
- Backing up files and software
- Distributing field updates to programs
- Aiding programmers—as a storage medium—during program preparation

While diskettes provide an advantageous low-cost medium for removable, permanent data storage, they have limited capacities. To save backup copies from a large, online disk, the system would require:

1. Many diskettes
2. A time-consuming interaction between an operator and the system when data must be copied or restored from multiple disks

Magnetic tape is conventionally used for this purpose because a single reel offers a large amount of storage. Because of the sequential nature of tape data storage, it is a suitable and efficient storage medium *only* for this type of backup.

Large-Volume Diskette

The IBM 4966 Diskette Magazine Unit is a more convenient and useful device with which to provide economical,

large-volume removable storage. This unit has a special diskette drive with two magazines, each of which contains ten diskettes plus three individual separate diskettes—a total of up to 23 diskettes. Users can remove both the magazines and the three individual diskettes.

The Diskette Magazine Unit is a large-capacity diskette unit that offers direct access to all data stored in the magazines or individual diskettes. The unit accommodates any type of diskette (single or double sided); storage volume can be as large as 1.2 megabytes per diskette—a maximum capacity of approximately 27 megabytes. The good performance characteristics of the unit insure that the system can use it effectively online and for backing up and copying online disks. System access time is 3 seconds to move from one diskette to the next, and a maximum of five seconds to reach any diskette. These timings total up to a 125,000 byte/second transfer rate after an average random access latency or delay of approximately 42 milliseconds.

The system requires about 10 seconds to completely load one, two-sided 1.2 megabyte diskette, and about five minutes to load the entire 23 diskette system containing 27 megabytes. To back up large data bases for either historical or error recovery purposes is a practical and economical procedure with the Diskette Magazine Unit.

User Attachment Features

As stated earlier, small computer applications require an integrated system of hardware, software, and maintenance; this often means the integration of special devices into the system. These special devices may be simple terminals or they may be complex analytical instruments, machines, or special hardware interfaces to other systems. Chapter 1 emphasizes how essential it is to integrate these devices into the hardware, software, and maintenance constituents of the system.

The Series/1 design pays particular attention to this requirement and provides the means to attach such hardware devices (Figure 17).

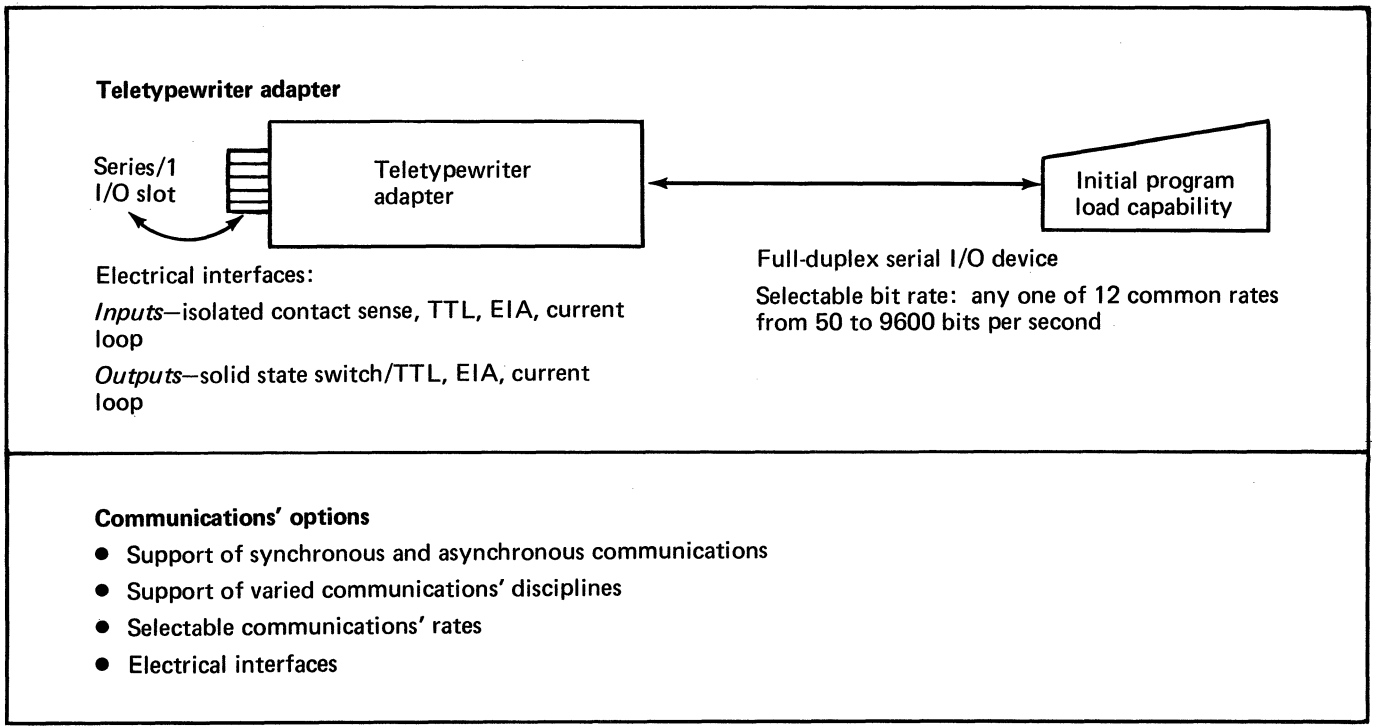


Figure 17. A subsystem can be flexibly configured to interface with a combination of analog and digital, input and output devices (1 of 4)

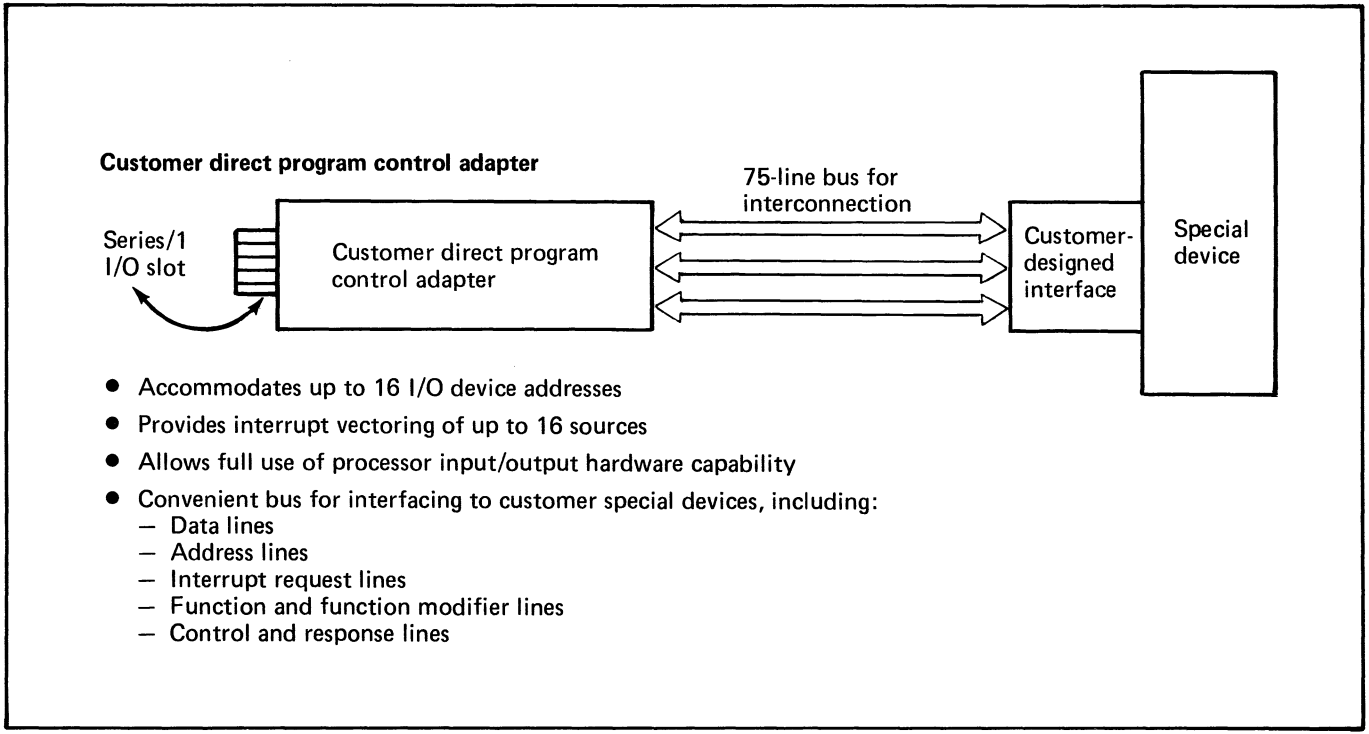
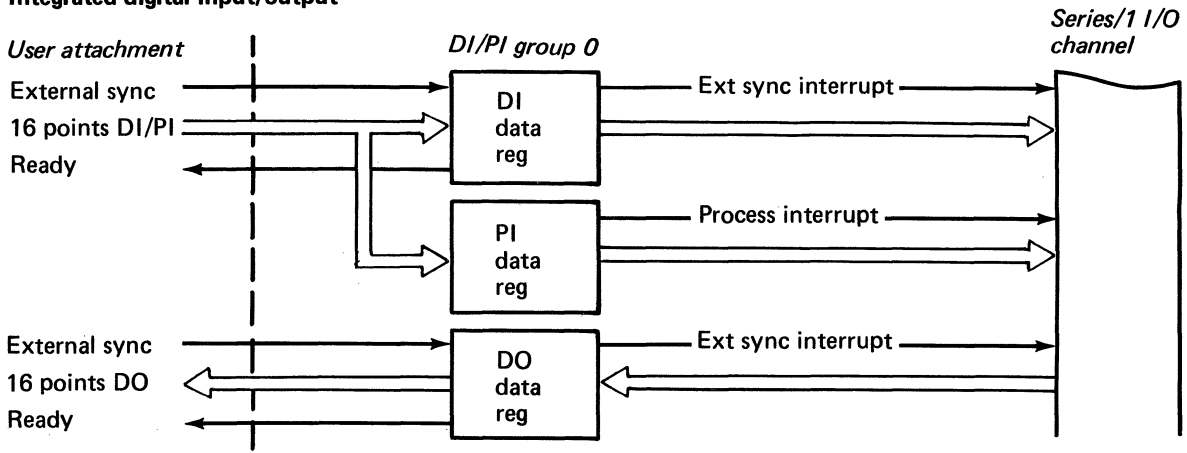


Figure 17. A subsystem can be flexibly configured to interface with a combination of analog and digital, input and output devices (2 of 4)

Integrated digital input/output



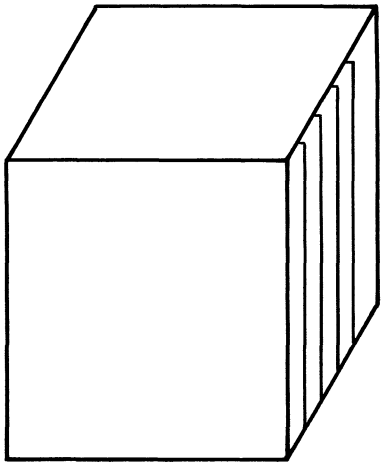
Two groups of 16 inputs and two groups of 16 outputs are available.

Ready and external sync signals permit easy synchronization of external devices.

The subsystem is supported by the Realtime Programming System through its interrupt capability and software support of digital input/output.

Figure 17. A subsystem can be flexibly configured to interface with a combination of analog and digital, input and output devices (3 of 4)

Sensor input/output unit



Software support for all analog and digital, input and output options

A subsystem with internal power supply supports up to eight input/output cards.

Cards or groups of cards which implement these functions:

- Analog output
- Analog to digital conversion with relay or solid state multiplexing
- Digital input/output

The subsystem can be configured--flexibly—to interface with sensors or devices requiring a combination of analog and digital, input and output signals.

Figure 17. A subsystem can be flexibly configured to interface with a combination of analog and digital, input and output devices (4 of 4)

Asynchronous Terminals

The Teletypewriter Adapter provides a means of attaching Teletype[1] ASR 33/35 or equivalent ASCII devices to the system. Many applications use such devices; this adapter provides a Series/1 compatibility with most of the terminals on the market today. Furthermore, original equipment manufacturers have designed many special-purpose, data entry devices that have been built to these same standard interface specifications. The Teletypewriter Adapter is the simplest user-device attachment because the software involved with this feature is minimal.

OEM Devices

The Customer Direct Program Control Adapter is much more general in its capability. IBM has designed this feature to perform direct program control for up to 16 user I/O devices. The interface fully integrates into the Series/1 which permits interrupt vectoring of all 16 sources. These user devices can then handle their associated interrupt response tasks like any other task using Control Program Support modules or the Realtime Programming System. The adapter is a convenient bus for control of arbitrary devices or subsystems. Interfacing to this adapter—which is similar to interfacing to any small computer or microprocessor, general purpose bus—is discussed in a later chapter.

The built-in, self-diagnostic capability of some of the Series/1 interfaces assists in the maintenance of systems which mix IBM-supplied devices and user-added devices. This is an important practical consideration for the Series/1 user.

The variety of communications' interfaces, which are useful for attaching to external devices and other systems, have already been discussed in this chapter.

The IBM Series/1 GPIB Adapter provides users with a means to connect to and control a variety of instrumentation and other devices which have been designed to be compatible with the Institute of Electrical and Electronic Engineers (IEEE) standard number 488. This standard is entitled "Digital

---

[1] Trademark of the Teletype Corp.

Interface for Programmable Instrumentations" and has been adopted by many device and instrumentation manufacturers. Consequently, Series/1 users can configure very flexible instrumentation systems without designing custom interfaces. Since the adapter operates off the cycle steal channel, the data rate is relatively high—a maximum of 65,000 bytes per second.

Sensor-Based Devices

Often the interconnection between a general purpose small computer and a customer device, subsystem, or process is through:

- Analog signals
- Digital signals
- Switches and relays
- Similar devices

Interfacing to these signals requires both digital input and output capability and a sensor-based input/output unit. Series/1 provides both the capability and the sensor.

Each digital input/output card incorporates 32 points of digital input and 32 points of digital output, together with external sync and ready lines for each 16-point group. These cards offer a convenient interface capability for devices whose inputs and outputs are in electronic digital registers. Interrupt-driven software support is equally convenient for this interface.

The sensor input/output unit has a separate, rack-mountable subsystem including a power supply. The subsystem supports:

- Digital input with process interrupt (either isolated or non-isolated)
- Digital output (non-isolated)
- Multiplexer-read relay input
- Multiplexer with solid state digital input
- Analog input with analog-to-digital converter
- Multirange amplifier for use with analog inputs and analog outputs

The system may be configured to support a specific mix of input and output types.

Software support of these features is standard in the Realtime Programming System. Users can, confidently, build their applications around the IBM-provided Series/1 hardware and software and then economically integrate their own I/O devices into the system.

Multiple Processors and a Shared Input/Output System

Certain critical applications require backup of the processor in case of system failure. In some of these applications, it is neither economical nor feasible to duplicate the input/output system; consequently, both the original processor—or any other processor backing up the original one—must be able to access the single input/output system. The Series/1 Two-Channel Switch control accomplishes this access by permitting the input/output channel to switch from one processor to another.

The switch can be operated manually or, through programming, on demand of the backup processor. The primary processor uses a 'dead man timer' counter in the interface to detect and signal failure. The primary processor periodically resets the timer. If it fails, the counter times-out and generates an interrupt to the backup processor which can then assume control of the input/output channel by commanding the Two-Channel Switch. In order to switch back to the primary processor, manual intervention is required.

Program Preparation Facilities

The Series/1 supports program preparation with either the Base Program Preparation Facilities (no operating system needed) or as background under control of the Realtime Programming System. In either case, source statements can be entered from interactive consoles or from diskettes prepared on offline key-to-disk units. An interactive or batch

text-editor facility permits updates and modifications to the source program.

Source language may be assembler language (including full macro capability), FORTRAN IV, PL/I, or COBOL. The PL/I language supports structured programming. The Series/1 PL/I provides a complete implementation of the standard language. Such an advanced high-level programming language has not previously been available on small computers.

FORTRAN, COBOL, and PL/I require extensive program preparation facilities supplied by the Realtime Programming System. After translation by the appropriate language translator, the object modules are combined with control blocks and tables to form a task set. This task set is used in realtime with the Realtime Programming System or for immediate execution in the batch stream. The system can build absolute or relocatable modules. Absolute modules enable a user-supplied supervisor to execute like dedicated applications.

Users control the storage of source and object code; thereby, they can take advantage of the disk storage units and the diskettes for program entry, storage backup, and distribution.

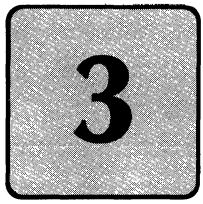
The Series/1 and Overall Application Needs

Although no single computer is a solution for all application and user problems, IBM has designed the Series/1 general purpose, small computer family with an architecture —an organization for hardware, software, maintenance, and support—which closely matches the needs listed in Chapter 1. The following characteristics, features, and specifics work together to make the Series/1 a long-term, viable solution for small computer applications:

- Support of cooperating tasks
- Excellent processor computational capability
- General interrupt and input/output system
- Integrated software support for small and large systems
- Substantial software preparation capability
- Ability to attach customer devices

- Availability of IBM documentation
- Customer engineering maintenance support in most countries around the world

In the succeeding chapters, each of the major features of the Series/1 is discussed in more detail to permit users to understand how each feature can be used to support their applications.



Processor Organization

The processor is the key element in a small computer system because it is responsible for:

- Control of data flow
- Interpretation and execution of instructions
- Response to external events
- Detection of internal events and errors
- Control of overall system integrity

Individual processors in the Series/1 share a common architecture but, in order to support applications of varying size and complexity, they may implement the overall architecture to differing degrees. This chapter discusses the overall Series/1 processor architecture and gives specific examples from existing processors. For complete details on any specific processor, the appropriate processor reference manual should be consulted.

Overall Flow of Information in the Series/1 Processors

Figure 18 shows the major elements in the Series/1 processors and the data paths connecting them. All

processors have four hardware levels of priority interrupt. To minimize overhead when responding to interrupts and switching from one level to another, the hardware implements a full set of general purpose and status registers for each priority level.

An important part of the processor is the interface to main storage. The processor must:

- Fetch instructions from storage
- Interpret these instructions
- Fetch data from main storage
- Carry out the instruction
- In some instances, return data to main storage

Specifically, consider the IBM 4955 processor, as an example. The processor bus is a 16-bit wide data path over which instructions, addresses, and data pass among the processor elements. The processor utilizes two registers for accessing main storage: the storage address register and the storage data register. The address of the next instruction to be fetched or the data item to be retrieved is transmitted along the processor bus to the storage address register.

Individual programs can address a maximum of 64K bytes (corresponding to a 16-bit address) at any one time. For processors without storage address translation, the maximum addressable main storage is 64K bytes and the address, generated by a program, is the physical address of the desired item in storage. For processor models with storage address translation, the program-generated address is hardware-mapped before physical memory is accessed. This allows main storage addressing of more than 64K bytes. The accessing of main storage is important and is discussed in detail in Chapter 4.

The storage data register contains whatever data, if any, is to be written into main storage and receives data and instructions being fetched from main storage. This register is 16-bits wide (the word length of the Series/1) and interfaces to the processor bus for transfer of the information to or from other registers.

All IBM Series/1 processor models are microprogrammed. That is, the system implements a set of micro instructions at the basic hardware level which control every register and data path in the processor. Processor instructions, on the other hand, are the machine instructions which manipulate data and perform input/output operations. A group of these instructions constitutes a program and is stored in main storage. The processor actually interprets and carries out these instructions by executing a series of micro instructions. Thus, each user instruction can be thought of as a "subroutine" made up of micro instructions, although this is transparent to the user.

Figure 18 shows an area denoted "Read only storage" where the system stores the microprograms for interpreting and executing instructions. The major advantages of a microprogrammed processor are:

1. A system can realize a very complete and general instruction set with reduced hardware cost penalties. A microprogrammed processor reduces the number of instructions required to perform a particular function; this, in turn, leads to a more compact system and more compact application software.
2. Without severe cost penalties, the user can add microprograms to the system for self-checking and self-diagnosis. This ability to detect errors is dependent upon a sophisticated software which can both detect and recover from errors.

The capability to manipulate the hardware elements at a very basic and general level means that—in order to isolate malfunctions—microprograms can be included to test arithmetic operations, logical operations, communications along data paths, and the many other hardware elements in a processor. This capability gives the processor a minimum-repair-time characteristic.

Figure 18 shows alternative storage address and data registers for the cycle stealing, input/output system. Cycle stealing involves the transfer of a series of data items to or from main storage. The transfer is initiated by the processor

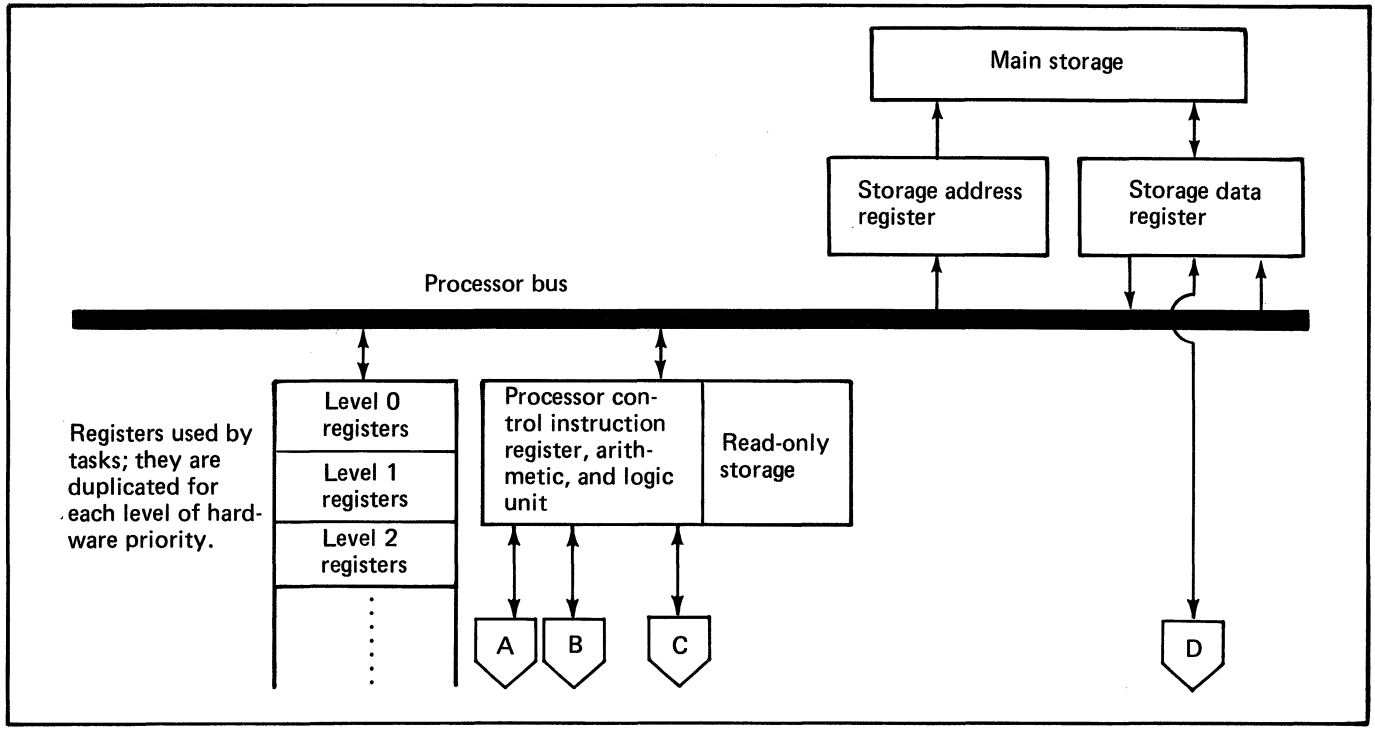


Figure 18. Overall data flow in the Series/1 processor (1 of 2)

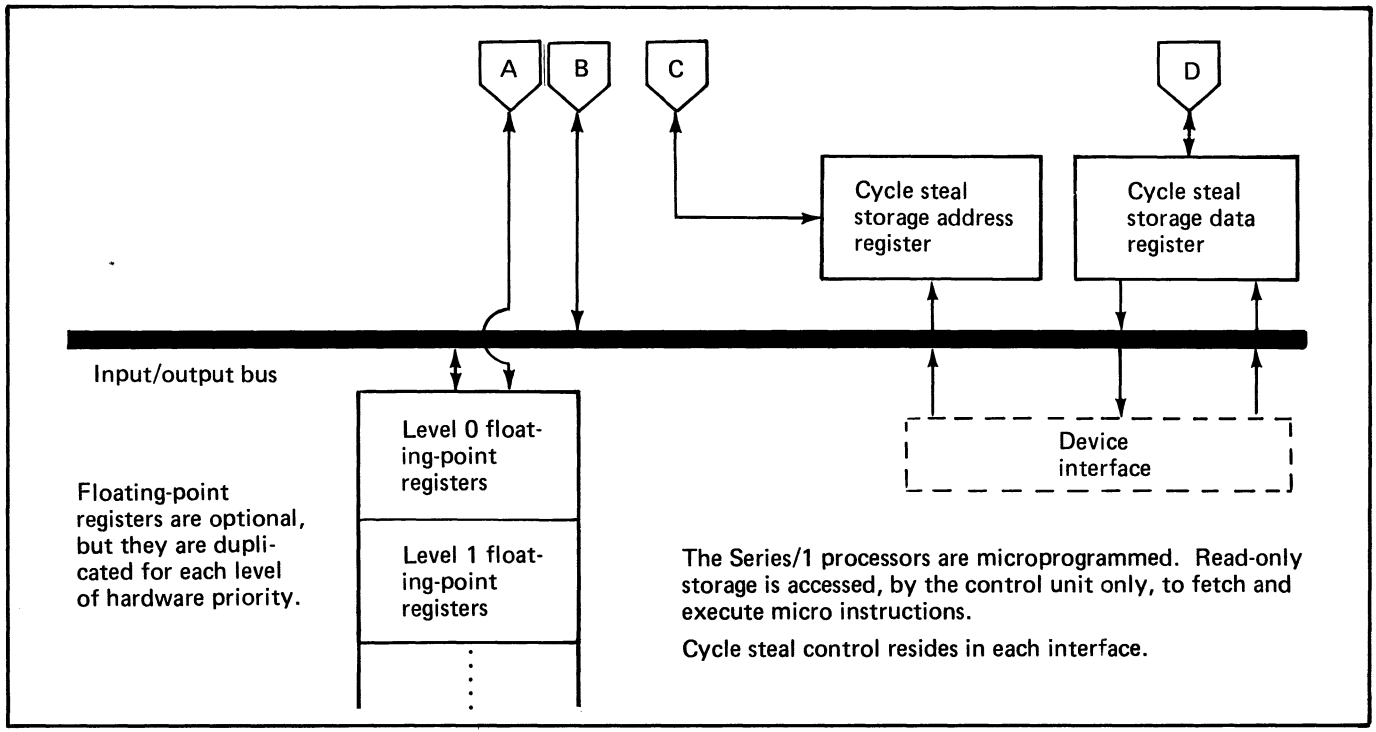


Figure 18. Overall data flow in the Series/1 processor (2 of 2)

but then proceeds, until complete, without further intervention. Thus, cycle stealing requires additional control hardware to handle the individual data transfers.

A major strength of the Series/1 architecture is the inclusion of the cycle steal control hardware in the individual device interfaces rather than in the processor itself. The microprocessor-controlled interface can perform the required control; the control can also recognize and handle the particular device characteristics. Furthermore, the system interfaces multiple devices on a cycle steal basis simply by providing the necessary control in the devices' interfaces themselves. The latter advantage is most apparent when many terminals or devices are interfaced: the load on the processor is minimal when the system performs input/output on a cycle steal basis rather than on a direct, program-control basis.

The cycle steal storage address and data register are used in the same way as the processor storage address and data registers, but the two kinds of registers are separate because input/output operations and processor operations can take place concurrently.

The main storage interface resolves any contention for main storage should both the processor and the cycle steal channel attempt simultaneous access. If this occurs, the channel is given priority, making the processor wait. Hence, the channel "steals" cycles from the processor. The input/output bus along which addresses and data are transferred is discussed in more detail in a later chapter.

The element in Figure 18 labeled "Processor control . . . arithmetic, and logic unit" is that portion of the processor responsible for controlling the sequence of operations in the processor, decoding instructions, fetching micro instructions from read-only storage, and carrying out the appropriate instruction. The instruction register holds the instruction fetched from main storage while it is decoded and executed. Not shown in Figure 18 are those registers used by the hardware but not accessible by user programs.

The arithmetic and logical unit provides hardware for arithmetic operations (except floating-point operations),

logical operations, and shifting operations. This unit is also the source of result-status information used in those instructions which test for:

- Result even or odd
- Carry of overflow conditions
- Result zero or negative

Since the arithmetic and logical unit is used by instructions on all priority levels, the indicators listed above are physically maintained in that level status register which is appropriate to the priority level on which the instruction was carried out. The level status register is one of the registers which is replicated for each priority level and is discussed later in this chapter. Operands to be processed by the arithmetic and logical unit are transferred across the common processor data bus under the control of the microprogrammed control unit. Processing is then carried out:

1. In the general purpose registers on each interrupt level
2. In processor registers which are not duplicated on each level but which do determine the level on which the processor operates
3. Via the interrupt mechanism which is responsible for controlling the response to internal and external events

Registers and Their Use by Tasks

Each hardware priority level contains a group of eleven 16-bit registers called the level status block as shown in Figure 19. The contents of these registers determine the "state" of the program executing on that level. In a sense, it is this information which the system must protect from change by other programs and which the system must restore if the registers on that level are used by another program. By using special instructions, the hardware design of the processor helps with this saving and restoring of the level status block contents. These instructions are illustrated later in this chapter.

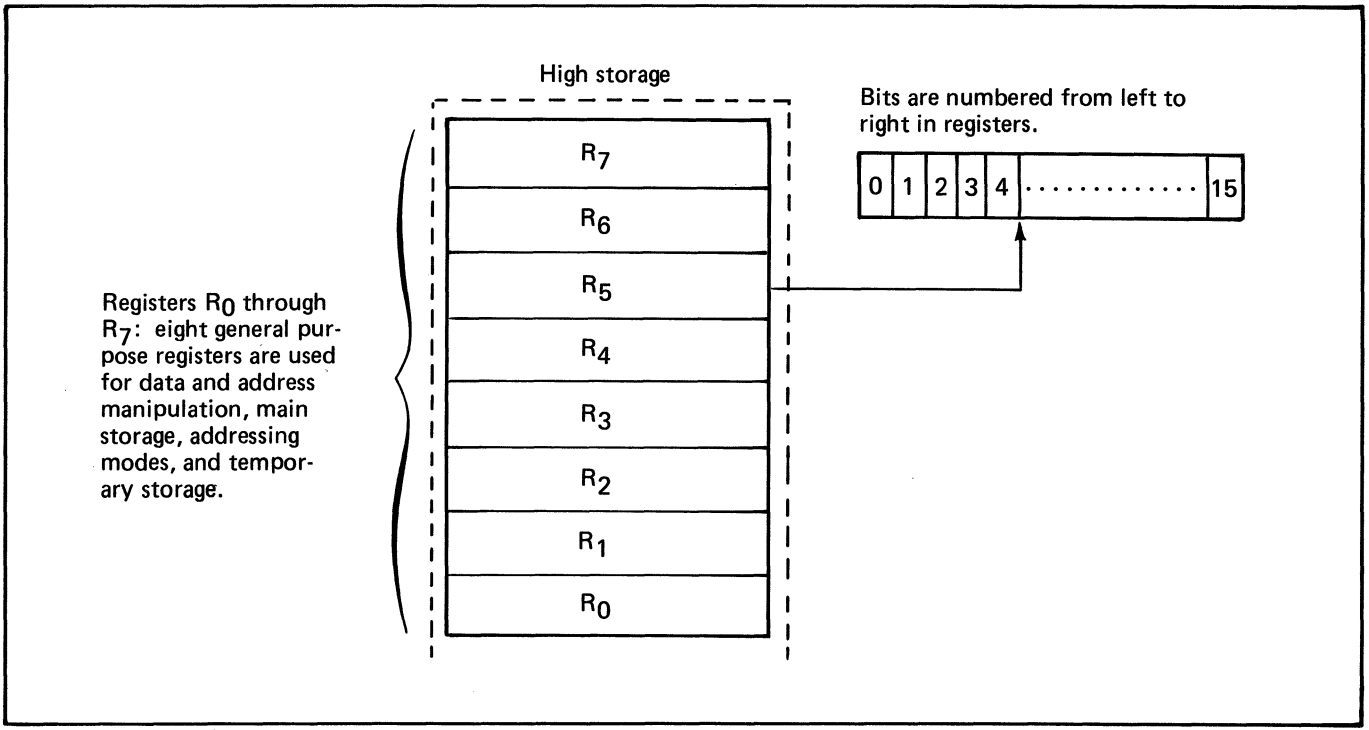
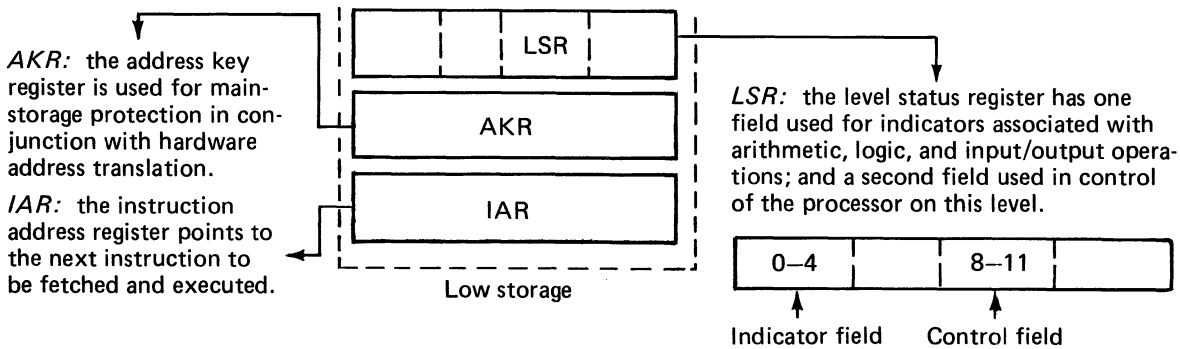


Figure 19. The level status block (1 of 2)



Four floating-point registers are optionally available but are not part of the level status block.

The eleven 16-bit registers shown here constitute the "state" information necessary to be saved and restored when programs are interrupted.

Special instructions, together with the design of the Series/1 architecture, provide commands to help software manipulate these registers as a single group; in turn, they exercise control over processor priority level and response to external interrupts.

Figure 19. The level status block (2 of 2)

In addition to the eleven registers in the level status block, four 64-bit registers for floating-point operations are optionally provided. The system uses this total of fifteen registers in three basic ways:

1. Data storage and manipulation
   a. Eight general registers
   b. Four floating-point registers
2. Addressing main storage
   a. Eight general registers
   b. Instruction address register
3. Task control
   a. Address key register
   b. Level status register

Other registers exist in the processor but are not referenced explicitly in instructions. For example, the arithmetic and logic unit shown in Figure 18 contains registers for temporary storage of data items during the execution of instructions using that unit. Note that, as shown in Figure 19, individual bits within any register are numbered from left to right starting with 0.

Storage and Manipulation of Data Types

Data storage and manipulation is one of the major tasks of a small computer; consequently, it is important that the processor support the variety of data types and data structures commonly used in applications. Data types supported by the Series/1 include:

Logical or Flag Variables

These are variables which take on the value of true or false and are denoted by 1 or 0 in storage. Stored 16 to a word, they may be tested and manipulated either a bit at a time or as a group. Instructions supporting this data type are discussed in Chapter 6.

Character Variables

These are variables whose length is eight bits and which are coded in some standard code format like ASCII or

EBCDIC. Characters occupy one byte in storage; a string of characters occupies sequential bytes in storage. Instructions for testing and manipulating individual characters and strings of characters are also discussed in Chapter 6. Input/output devices help test and manipulate characters—especially communications' interfaces as discussed in Chapter 8.

Unsigned and Signed Numbers of Various Precisions

Processor instructions support both signed and unsigned numbers of single precision (16 bits), and double precision (32 bits). Formats for the numbers are shown in Figure 20 together with the range of numbers permitted in each.

Note: Changing from double precision to single precision or vice versa is a straightforward operation with these formats. For example: adding a word whose bits are all zero and appending it to any single-precision variable, automatically extends it to double precision. This is so because bit 0 (the leftmost bit) is *not* treated as a sign bit in the second word of double-precision variables. With the aid of the level status register, the system performs arithmetic and logical operations in a straightforward manner on either signed or unsigned variables—including detection of exception conditions—as indicated below.

Not only are the signed and unsigned numbers useful in applications, but the hardware also supports the use of higher-precision variables. Certain instructions allow the addition or subtraction of multi-word operands—taking into account any carry from similar operations on previous words, and setting the indicators to reflect the multi-word result for use in the next stage of the calculation.

Floating-Point Numbers with Two Precisions

Single- and double-precision, floating-point numbers—which occupy two words or four words—are supported by the optional floating-point processor and are illustrated in Figure 20. The format for these variables is identical to that used in the IBM System/360 and System/370 computers: an eight-bit exponent considered to be a power of 16 in the range -64 to +63 and a fractional part, with only the length

Unsigned numbers: positive numbers only

- Byte length—range from 0 to 2^8-1
- Single precision—range from 0 to $2^{16}-1$
- Double precision—range from 0 to $2^{32}-1$
- The instruction set supports addition and subtraction of unsigned numbers
- Extension of precision involves addition of 0s to the most significant byte or word

Unsigned, multiple precision

- Any number of words
- Arithmetic operations are programmed as a series of operations using special instructions to include carry resulting from the previous steps

Signed, byte-length numbers

- 8-bit numbers
- Twos complement form
- Bit 0 represents the sign
- Numbers range from -2^7 to 2^7-1
- Byte in storage or the least significant half of a register (bits 8 through 15)

Signed, single-precision numbers

- 16-bit numbers
- Twos complement form
- Bit 0 represents the sign
- Numbers range from -2^{15} to $2^{15}-1$
- Word in storage or registers

Figure 20. Floating-point numbers (1 of 3)

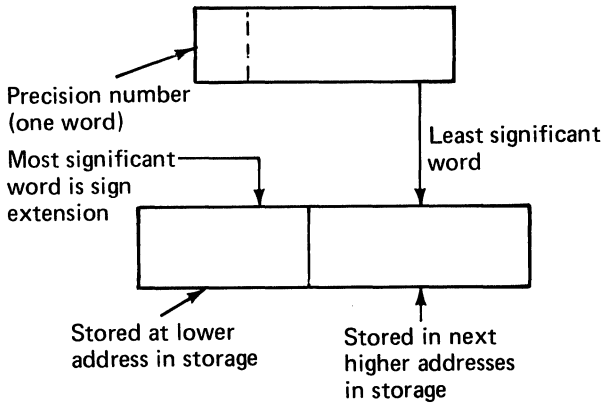
of the fractional part differing between single and double precision. This, again, facilitates conversion back and forth between the two precisions.

Operations on the various formats of numbers require:

- Instructions for addition, subtraction, multiplication, and other arithmetic operations

Extension of precision

Extending the precision of twos-complement, signed numbers involves extending the sign (0s if positive and 1s if negative) two additional bits to the left.



Signed, double-precision numbers

- 32-bit numbers
- Twos complement form
- Bit 0 of the first word represents the sign
- Numbers range from -2^{31} to $2^{31}-1$
- Two successive words in storage or two successive registers
- Most significant part of the word is stored in the lower-numbered register or storage location
- Least significant part of the word is stored in the higher-numbered register
- Bit 0 in second word is part of the value and is not treated in a special manner as is bit 0 in the first word (the sign bit)

Signed, multiple-precision numbers

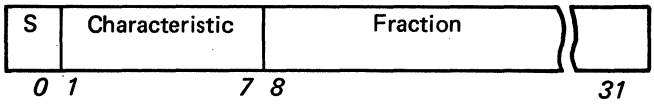
- Any number of words in length
- Arithmetic operations are programmed as a series of operations using special instructions which include carry and overflow resulting from the previous steps

Figure 20. Floating-point numbers (2 of 3)

Single-precision, floating-point numbers

32-bit numbers in the IBM System/370 format; they are stored in the most significant 32 bits of the 64-bit, floating-point register or in two successive storage words.

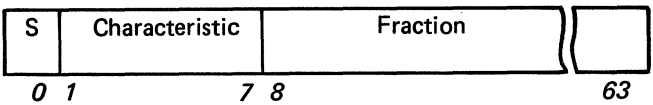
Short floating-point number—used for single precision



Double precision, floating-point format

64-bit numbers in the IBM System/370 format; they are stored in the 64-bit, floating-point register or in four successive words in main storage.

Long floating-point number—used for double precision



Standard, floating-point format

- Sign: stored in bit 0
- Characteristic: seven-bit number indicating a power of 16 and a stored excess of 64. Exponents range from -64 through +63 and correspond approximately to the range 10^{-76} to 10^{76}.

Fraction

- An unsigned number between 0 and 1. Floating-point numbers are *normalized.* This means that the leading hexadecimal digit of the fraction is nonzero. Hence, the fraction is actually between 1/16 and 1 at all times.
- The 24-bit, single-precision fraction corresponds to about 7 significant figures; the 56-bit, double-precision fraction corresponds to about 16 significant figures

Figure 20. Floating-point numbers (3 of 3)

- Testing and comparing instructions
- A method for detecting special results such as arithmetic overflow

The level status register contains the last listed information in its indicator bits which provide sufficient information concerning the operations' result to permit full use of all number formats. Figure 21 defines the portion of the level status register used for this purpose. As shown there, each operation (arithmetic, logical, compare, shift, and complement) sets the even, carry, overflow, negative, and zero indicators in the level status register in a way appropriate to the operation. Instructions for testing and branching on each condition are provided and discussed in Chapter 6.

Of particular interest is the overflow condition which can occur in numerical operations. In both signed and unsigned operations, the system detects and obtains—from the bits in the level status register—the true value of any result which is too large to be contained in a register or register pair. For example: overflow, resulting from the addition of a signed number-pair, both sets the overflow indicator and puts the extra bit in the correct result in the carry indicator. The system provides similar support of unsigned, numerical operations.

All of the various data types and number formats are fully supported in all programming languages available on the Series/1. Because of the extensive hardware support of these formats by the processor, implementation of languages in which numerical processing is extensive (such as FORTRAN and PL/I) is especially efficient.

Processor States and the Interrupt System

IBM has designed the Series/1 processor to be responsive to external and internal events, especially the detection and recovery of errors. To accomplish this, the processor has a number of states or conditions, in each of which the system performs different functions. Figure 22 shows these states and the transitions that may occur among them. The non-running states (power-off, stop, wait, and load) are

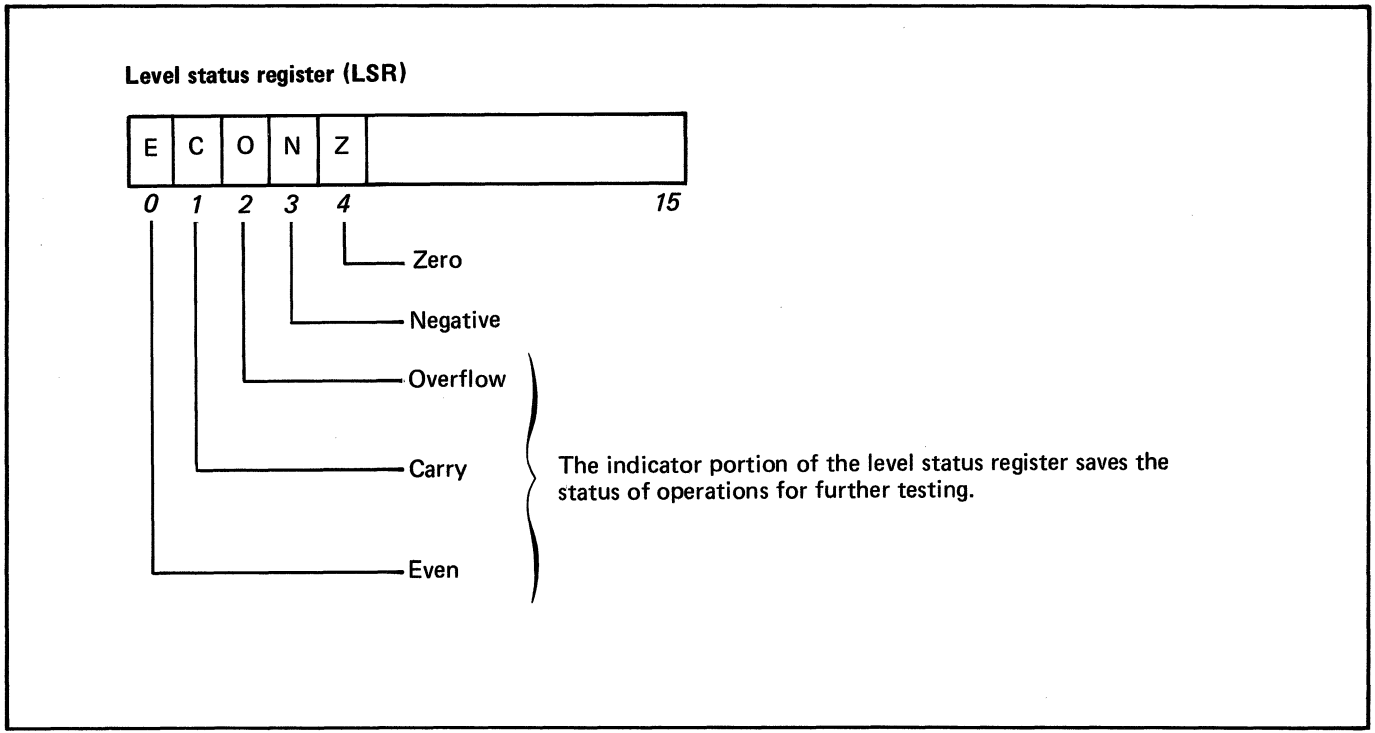


Figure 21. Indicator set in the level status register (1 of 2)

Arithmetic operations:	The even, negative, and zero conditions are set to correspond to the result of the operation.
Signed numbers:	1. Use overflow to indicate a result too large to be represented 2. If overflow is indicated, carry is used to contain the extra bit. Carry combined with the result register is the correct result of an arithmetic addition or subtraction which overflows.
Unsigned numbers:	Use carry to indicate a result which cannot be represented. Carry combined with the result is the correct result of an arithmetic addition or subtraction. In the latter case, the result is in twos-complement form, even though the original numbers were unsigned.
Floating-point numbers:	1. Use overflow to indicate a result too large to be represented, and for underflow (too small a result to be represented), and division by zero 2. Carry is used to indicate division by zero, and to indicate underflow. Hence, all three indicators must be checked to determine which conditions occurred.
Input/output operations:	The three indicators (even, overflow, and carry) are used as a three-bit condition code which is set after each input/output operation. This is discussed further in the section concerning these operations.
Non-arithmetic operations:	The indicators are used for special purposes by various instruction classes. The appropriate processor reference manuals should be consulted for specific conditions under which indicators are either changed or unaffected by each instruction.

Figure 21. Indicator set in the level status register (2 of 2)

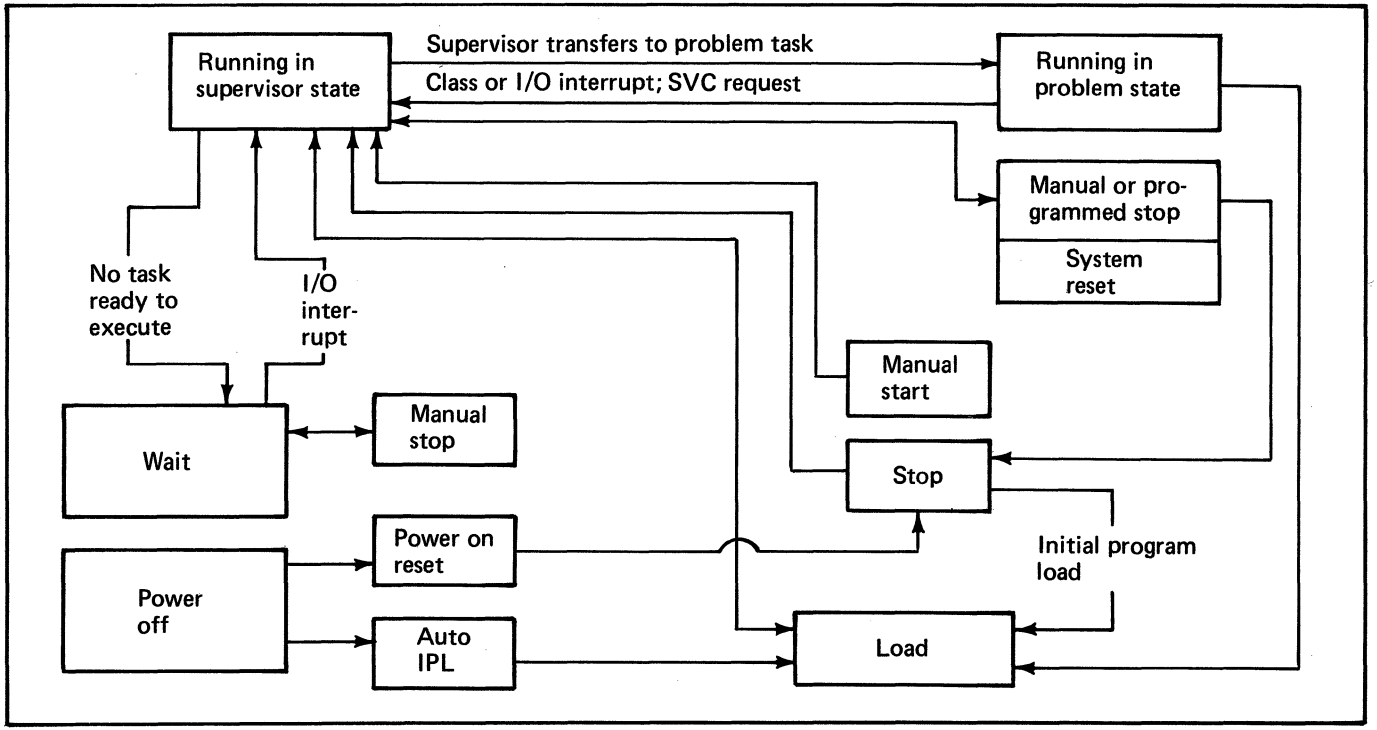


Figure 22. Basic processor states and the transitions among them

concerned with start-up of the processor, initialization of programs, and other functions. Their definitions are:

Initial Program Load (IPL) State

The system provides an initial program load function to: (1) read an IPL record (set of instructions) from an external storage media, and (2) automatically execute a start-up program. An IPL record reads into storage from a local I/O device or host system. The I/O attachments for the desired IPL sources are prewired at installation time. The user can wire two local sources—primary and alternate—and select either one by using the IPL source switch on the console.

IPL can be started by three methods:

1. Manually, by pressing the load key on the console
2. Automatically, after a power-on condition
3. Automatically, when a host system sends a signal. The host system can be connected through a communications' adapter.

The user selects the automatic power-on IPL by a mode switch on the console. When the mode switch is in the auto IPL position, IPL occurs whenever power turns on (either initially or after a power failure). Auto IPL is useful for unattended systems. A user can initiate a manual IPL at any time by pressing the load key on the console (even when in the run state). The mode switch has no effect on the manual IPL. For auto IPL and manual IPL, the local IPL source (primary or alternate) is selected.

IPL from a host system can occur at any time; the host system initiates it. The system transfers the IPL record through the host-system device—the communications' adapter, for example. When an automatic IPL occurs, one bit (bit 13) in the processor status word is turned on so that system software can detect this condition. If the source of the IPL is either a manual command or the host computer system, this bit sets to zero. Upon successful completion of an IPL, an I/O interrupt occurs, forcing the processor to enter the supervisor state and to begin execution at address zero. The summary mask and all priority interrupt levels in the mask register are enabled.

The system normally returns control to a routine in the operating system which reloads all pertinent system tables and sets up the system to respond properly to interrupts.

Stop State

The stop state is entered when:

1. The stop key on the programmer console is pressed
2. The stop instruction is executed, and
   a. The mode switch on the basic console is in the diagnostic position, and
   b. The optional programmer console is installed
3. An address-compare occurs and the rate control on the programmer console is in the stop on address position
4. An instruction has completed execution and the rate control on the programmer console is in the instruction step position
5. An error occurs and the error control on the programmer console is in the stop on error position
6. The reset key on the programmer console is pressed
7. Power-on reset occurs

The processor exits the stop state when:

1. The load key on the basic console is pressed
2. The start key on the programmer console is pressed. When the start key is pressed, the processor returns to the state that was exited before entering the stop state. If the run state is entered, one instruction is executed before interrupts are accepted by the processor. If the stop state was entered because of a reset (power-on reset or reset key), pressing the start key causes program execution to begin on level zero with the instruction in location zero of main storage. If the stop state was entered because of an error, with the stop on error switch turned on, a system reset must occur to clear the error condition.

Wait State

The processor enters the wait state when it leaves the current priority level and there is no task waiting on any

level. In this case, there is no task to execute and the processor waits without executing instructions until an interrupt occurs. While the processor is in the wait state:

1. The wait light on the basic console is on, and
2. The processor accepts interrupts under control of the system mask register and the summary mask (as defined by the LSR of the last active level)

The processor exits the wait state when:

1. The stop key on the programmer console is pressed
2. The reset key on the programmer console is pressed
3. An I/O interrupt is accepted (the level must be enabled)
4. A class interrupt occurs

Load State

The processor enters the load state when initial program load (IPL) begins. This occurs:

1. When the load key on the basic console is pressed
2. After a power-on reset if the mode switch is in the auto IPL position
3. When an IPL signal is received from a host system

While the processor is in the load state, the load light on the basic console is on. The processor exits the load state and enters the run state upon successful completion of the IPL.

Supervisor and Problem States

The two running states, supervisor state and problem state, are similar in that instructions are executed in each. They differ because the system imposes a restriction on those instructions that can be legally executed in the problem state. The purpose of this restriction is to provide a hardware environment for an operating system which can be fully protected against application program errors. That is, the system isolates one task from another by preventing the application program from directly executing:

- Input/output instructions
- Instructions associated with the management of registers
- Functions pertinent to the overall system

The supervisor state is normally restricted to the operating system and its subprograms. Of course, in a customized operating system or in an application with a special purpose software system, the supervisor state may be utilized in any way pertinent to the application.

The supervisor state is entered whenever either an input/output interrupt or a class interrupt occurs. This procedure is explained in the next section of this chapter.

In addition, an initial program load causes the processor to enter the supervisory state. In this manner, the system is able to control tasks and initialize an operating system properly after an event occurs. The processor leaves the supervisory state by executing a privileged instruction which sets the state bit in the level status register.

Effect of Interrupts on the Processor State

Responding to internal and external events requires:

- Recognition of an interrupt
- Control over priority of the interrupt
- Transfer of control to a responding program
- Eventual return of the system to its pre-interrupt status

Figure 23 shows the two general types of interrupts and the response that is made to them.

Input/Output Interrupts

The first type of interrupts comes from external sources and is labeled input/output interrupts. Any external interrupt—whether it is from a device or a process signal—falls into this category. Each source has a unique identification which is used to enter an interrupt branching table in main storage to find the location of the response routine. When recognized, the source places its identification number in general register

seven on the level of the interrupt. Since an I/O interrupt always occurs on a priority level higher than the currently executing program, that higher level must be idle; consequently, the contents of the registers are not meaningful at the time of the interrupt. Therefore, it is not necessary to save, and later restore, register seven's contents prior to using it for the device identification.

The processor uses the device identification number to find a pointer in the interrupt branching table, which in turn locates an area in main storage called the device data block. This buffer contains the address of the response routine which is loaded into the instruction address register of the proper level to start the response routine. Unique device description parameters are also automatically loaded.

Internal or Class Interrupts

The second type of interrupts are internal or class interrupts. They come from several sources but, in general, are related to the task executing on the current priority level of the processor. Consequently, these interrupts are responded to on the same priority level. Because a task is currently executing on that level, all registers are in use; the processor must save the registers' contents before transferring control to an interrupt response routine. The processor cooperates with interrupt response software by making further interrupts during examination of the critical registers. This procedure is illustrated in Figure 23 where the particular type of internal interrupt shown is used to select two addresses from the table:

1. The address of the interrupt response routine
2. The address of a save area for the level status block (i.e., the eleven user registers defined in Figure 19)

Following detection of the interrupt, the registers are saved, the processor enters the supervisor state, and the system transfers control to the starting instruction of the response routine. This routine can query the processor status word to further identify the source of the interrupt.

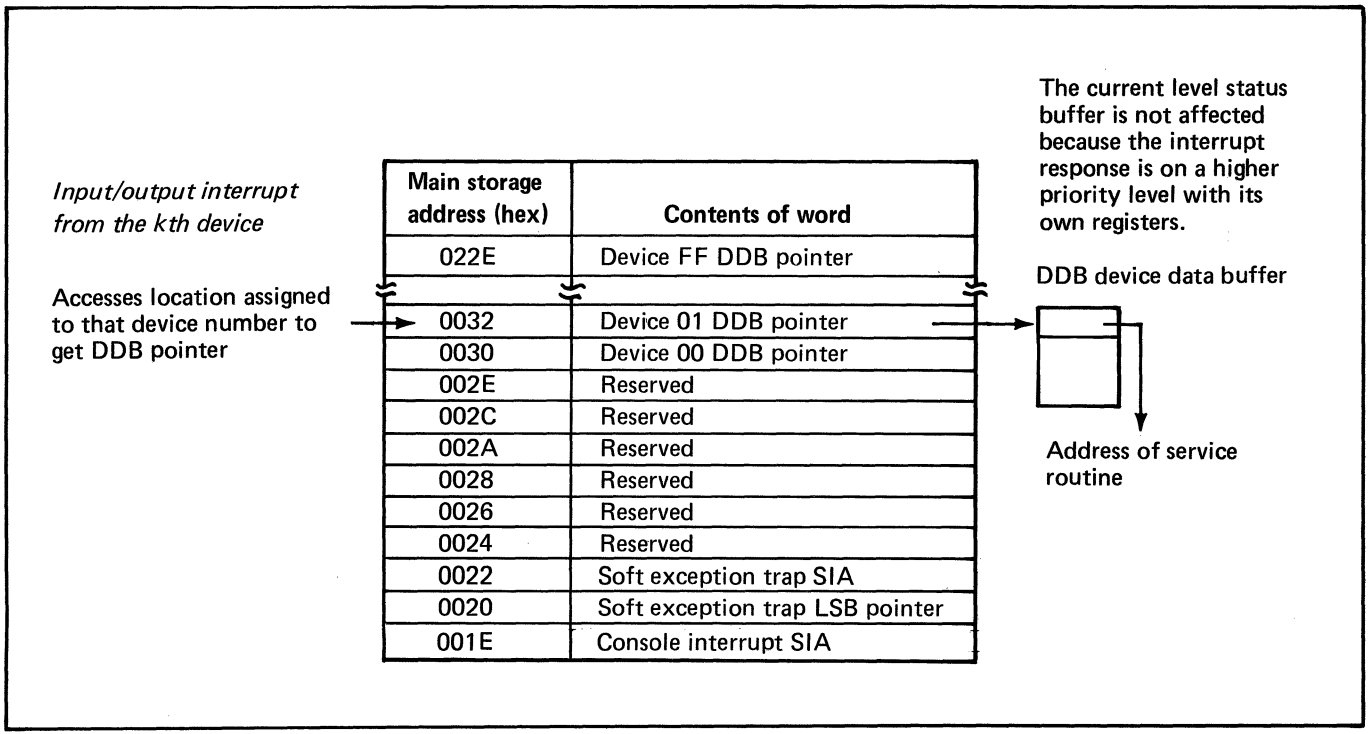


Figure 23. Input/output and class interrupts and the response of the processor (1 of 2)

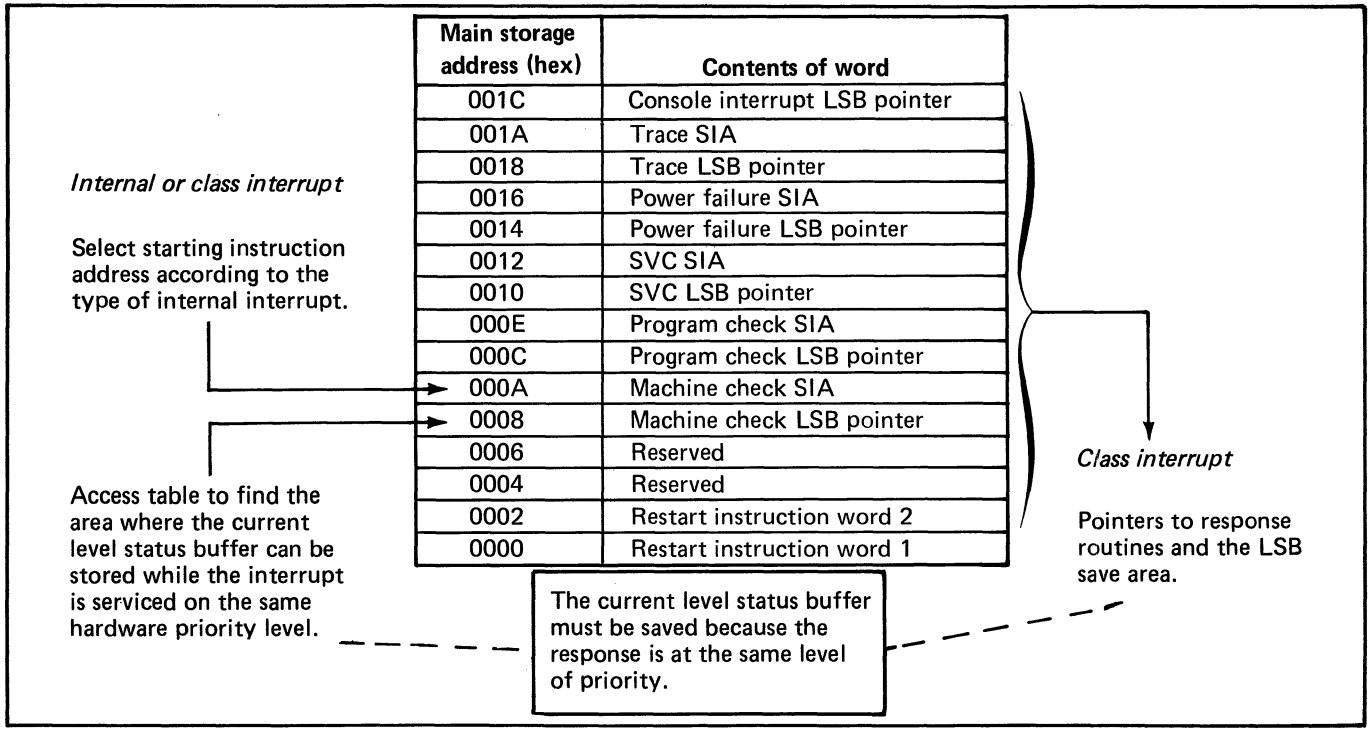


Figure 23. Input/output and class interrupts and the response of the processor (2 of 2)

Different Responses to the Two Types of Interrupts

The major difference between input/output interrupts and class interrupts is that class interrupts are responded to on the same priority level while input/output interrupts are recognized only if they are on a higher priority level. This difference is illustrated further in Figure 24 which describes the effects of priority. Note that once the processor responds to an interrupt on a given level, it remains on that level until it deliberately leaves by executing a privileged instruction such as LEX (Level Exit). As noted previously, class interrupts are responded to on the same level. Following completion of the interrupt response routine, control is normally returned to the interrupted task by restoring the saved registers in the level status block. The special instruction for this restoration is SELB (Set Level Block). Following recognition of a class interrupt, it is necessary to:

1. Identify the specific source of the event, and
2. Perform the proper error recovery procedure

The processor provides the processor status word (a system register whose contents are defined in Figure 25) which uses a set of flags to indicate the specific internal error or event that occurred. In an error situation, the response routine examines the processor status word and carries out an error recovery procedure.

The specific error recovery procedure depends on the task, the kind of error, and the current circumstances of the system; the recovery procedure will vary from application to application. It is important to remember that a user can design error recovery into an application thereby producing robust, non-sensitive, application programs. Certain high-level languages like PL/I permit the application programmer to specify the response to some internal events—a particular convenience, and one of the major advantages of PL/I as a programming language.

The Series/1 operating system also permits the user to attach tasks to these internal events. Thus, without giving up the generality of the operating system, users can conveniently control the error recovery of their applications.

Class Interrupts in the Use of Stacks

As an example of the integration of hardware and software, consider the stack operations of the Series/1 processor. The processor takes advantage of the internal interrupt systems to make often-used software more efficient. In current operating systems and applications, it is often important to use a stack data structure for control of re-entrant software, allocation of storage areas, storage of data, and other operations.

The processing unit offers two types of stacking facilities:

Data Stacking. This facility provides an efficient and simple way to handle last-in-first-out (LIFO) queues of data items and/or parameters in main stack elements. For a given queue (or stack), each element is one-, two-, or four-bytes wide. The system incorporates instructions for each element size (byte, word or doubleword) to:

1. Add an element to the stack (register to storage). This is popularly called "pushing" the element onto the stack.
2. Delete the last entered element from the stack. This is popularly called "popping" the stack.

Linkage Stacking. This facility provides an easy method for linking subroutines to a calling program. The system uses a word stack for saving and restoring the status of general registers, and for allocating dynamic work areas. The Store Multiple (STM) instruction stores the contents of the registers in the stack, and reserves a designated number of words in the stack as a work area. The Load Multiple and Branch (LMB) instruction reloads the registers, releases the stack elements, and causes a branch, via register 7, back to the calling program.

Data Stacking Description

Any contiguous area of main storage can be defined as a stack. Each stack is defined by a stack control block.

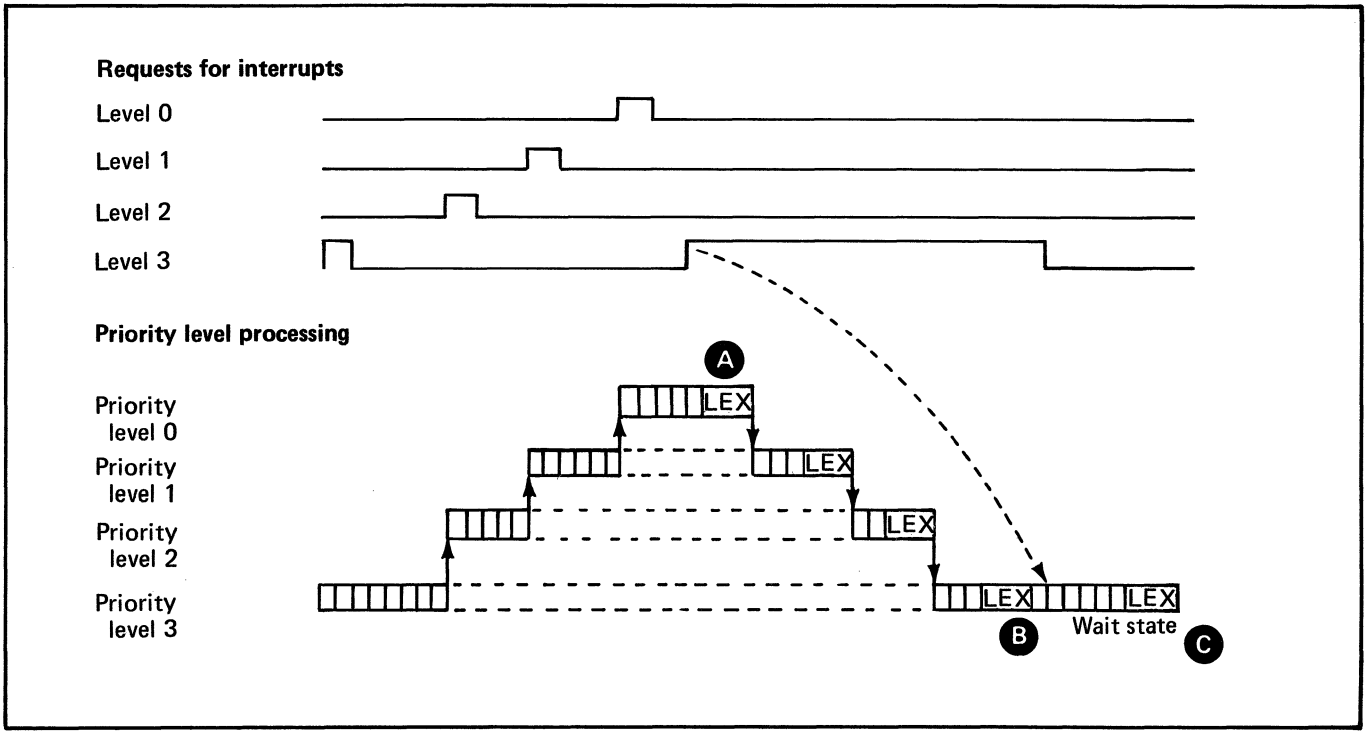


Figure 24. Multilevel priority interrupt response (1 of 2)

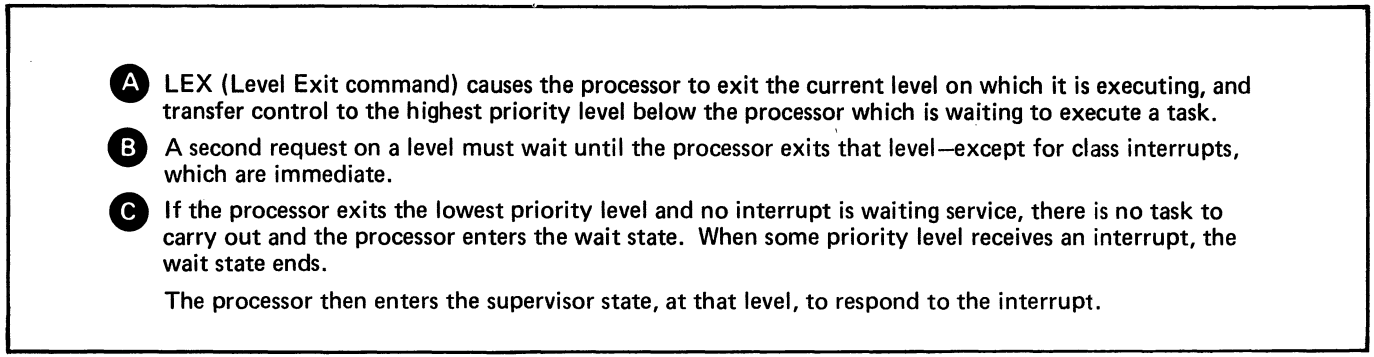


Figure 24. Multilevel priority interrupt response (2 of 2)

Use of the processor status word

The processor status word (PSW) is used to record error or exception conditions in the system that may prevent further processing. It also contains certain status flags related to error recovery.

The PSW is contained in a 16-bit register with the following bit representation:

Bit	*Condition*	*Class interrupt*	*Remarks*
00	Specification check	Program check	
01	Invalid storage address	Program check	
02	Privilege violate	Program check	
03	Protect check	Program check	
04	Invalid function	Program check or soft exception trap	
05	Floating-point exception	Soft exception trap	
06	Stack exception	Soft exception trap	
07	Not used		Always zero
08	Storage parity check	Machine check	
09	Not used		Always zero
10	CPU control check	Machine check	
11	I/O check	Machine check	
12	Sequence indicator	None	Status flag
13	Auto-IPL	None	Status flag
14	Translator enabled	None	Status flag
15	Power/thermal warning	Power/thermal	

Error or exception conditions recorded in the processor status word cause the following four class interrupts:

1. Machine check, caused by a hardware error
2. Program check, caused by a programming error
3. Power/thermal warning, caused by a power or thermal irregularity
4. Soft exception trap, caused by software

Other class interrupts not recorded in the processor status word are:

1. Supervisor call, caused by execution of an SVC instruction
2. Trace, caused by instruction execution (trace enabled in the current LSR)
3. Console, caused by a console interrupt when the optional programmer console is installed

Figure 25. The processor status word (1 of 4)

Examples of definitions of processor status word flags and conditions

Program check conditions

Bit 00 Specification check. Set to one if the storage address violates the boundary requirements of the specified data type.

Bit 01 Invalid storage address. Set to one when an attempt is made to access a storage address outside the storage size of the system. This can occur on an instruction fetch, an operand fetch, or an operand store if the system is using a translator and the segment register is declared invalid.

Bit 02 Privilege violate. Set to one when a privileged instruction is attempted in the problem state (supervisor state bit in the level status register is not on).

Bit 03 Protect check. In the problem state, this bit is set to one when (1) an instruction is fetched from a storage area not assigned to the current operation, (2) the instruction attempts to access a main storage operand in a storage area not assigned to the current operation, or (3) the instruction attempts to change a main storage operand in violation of the read-only control.

Program check or soft exception trap condition

Bit 04 Invalid function: Set to one by one of the following conditions:

1. An illegal operation code or function combination
2. The processor attempts to execute an instruction associated with an uninstalled feature

Figure 25. The processor status word (2 of 4)

Figure 26 shows a data stack and its associated stack control block. The user must align stack control blocks on a word boundary. The words in the stack control block are used as follows:

High Limit Address (HLA). This word contains the address of the first byte beyond the area being used for the stack. All data in the stack has a lower address than the contents of the HLA. Note that the HLA points to the first byte beyond the bottom of an empty stack.

Soft exception trap condition

Bit 05 Floating-point exception. Set to one when an exception condition is detected by the optional floating-point processor. The arithmetic indicators (carry, even, and overflow) define the specific condition.

Bit 06 Stack exception. Set to one when an attempt has been made to remove (pop) an operand from an empty main storage stack or enter (push) an operand into a full main storage stack. A stack exception also occurs when the stack cannot contain the number of words to be stored by a Store Multiple (STM) instruction.

Machine check conditions

Bit 08 Storage parity. Set to one when a parity error has been detected on data being read out of storage by the processor. This error may occur when accessing a storage location that has not been validated since power on.

Bit 10 CPU control check. A control check will occur if no levels are active but execution is continuing. This is a machine-wide error.

Bit 11 I/O check. Set to one when a hardware error has occurred on the I/O interface that may prevent further communication with any I/O device.

PSW bit 12 (sequence indicator) is a zero if the error occurred during an Operate I/O instruction and is set to one if the error occurred during a non-DPC transfer.

Figure 25. The processor status word (3 of 4)

Low Limit Address (LLA). This word designates the lowest storage location that can be used for a stack element. Note that the LLA points to the top of a stack.

Top Element Address (TEA). This word points to the stack element that is currently on top of the stack. For empty stacks, the TEA points to the same location as the high limit address (HLA).

Notes:

1. For word, doubleword, and register block operations, the high limit address, low limit address, and top element

Status flags

Bit 12 Sequence indicator. This bit reflects the last I/O interface sequence to occur after an I/O check.

Bit 13 Auto IPL. Set to one by hardware when an automatic IPL occurs. Set to zero by a power-on reset when the mode switch is not in auto-IPL, by pressing the load key, or by a host-system IPL.

Bit 14 Translator enabled. When the Storage Address Relocation Translator Feature is installed this bit is set to one or zero as follows:

1. Set to one (enabled)
   a. An Enable (EN) instruction is executed with bit 12 of the instruction word set to zero and bit 14 set to one
2. Set to zero (disabled)
   a. A Disable (DIS) instruction is executed with bit 14 of the instruction word set to one
   b. An Enable (EN) instruction is executed with bit 12 of the instruction word set to one
   c. A processor reset (power-on reset, check restart, IPL, or system reset key)

Power/thermal warning condition

Bit 15 Power warning and thermal warning. Set to one when these conditions occur. The power/thermal class interrupt is controlled by the summary mask.

Figure 25. The processor status word (4 of 4)

address must all contain an even numbered address to insure data alignment on a word boundary.

2. The high limit address and low limit address define a contiguous range of addresses. These addresses must not cross the 64K-byte boundary because that action causes storage to wrap around, i.e. correspond to addresses at the beginning of storage. Figure 27 shows how elements are pushed into and popped from a stack. Note that each push operation always places an element at a lower address in the stack than the preceding element.

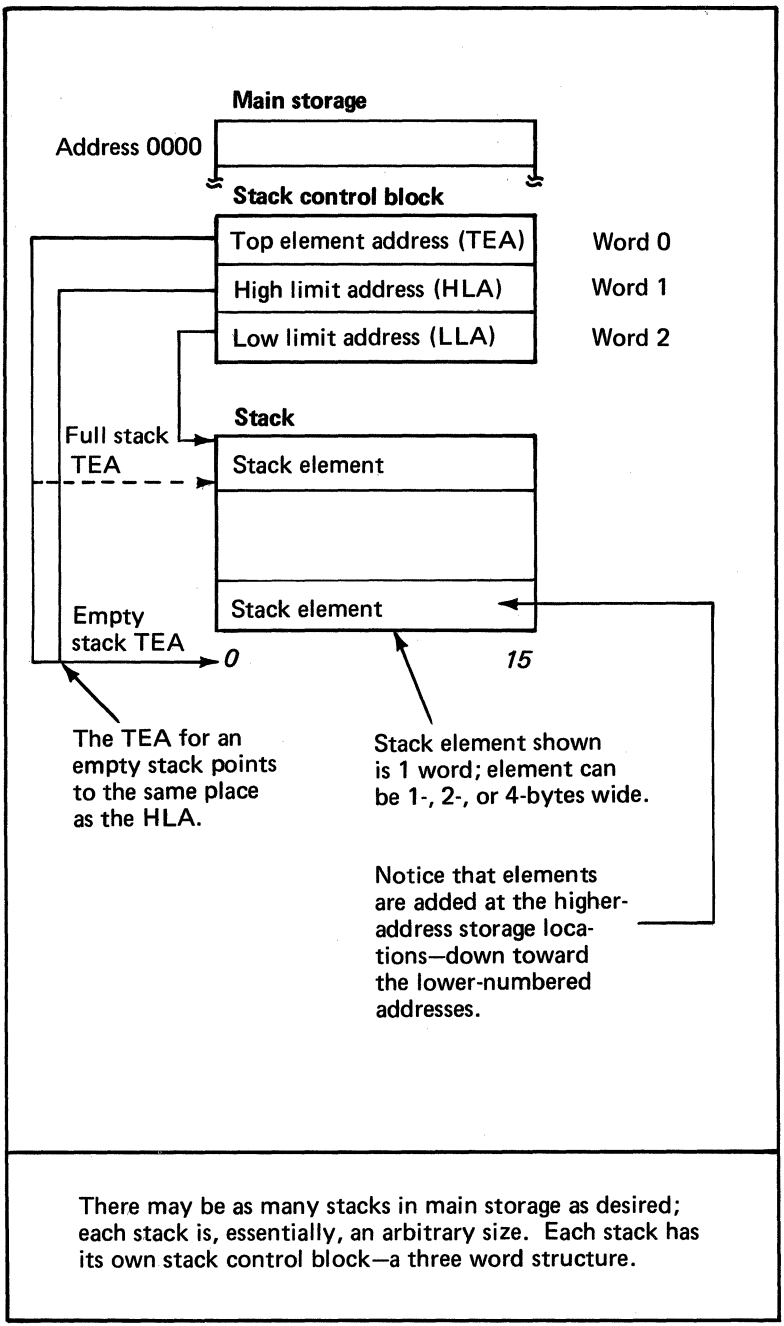
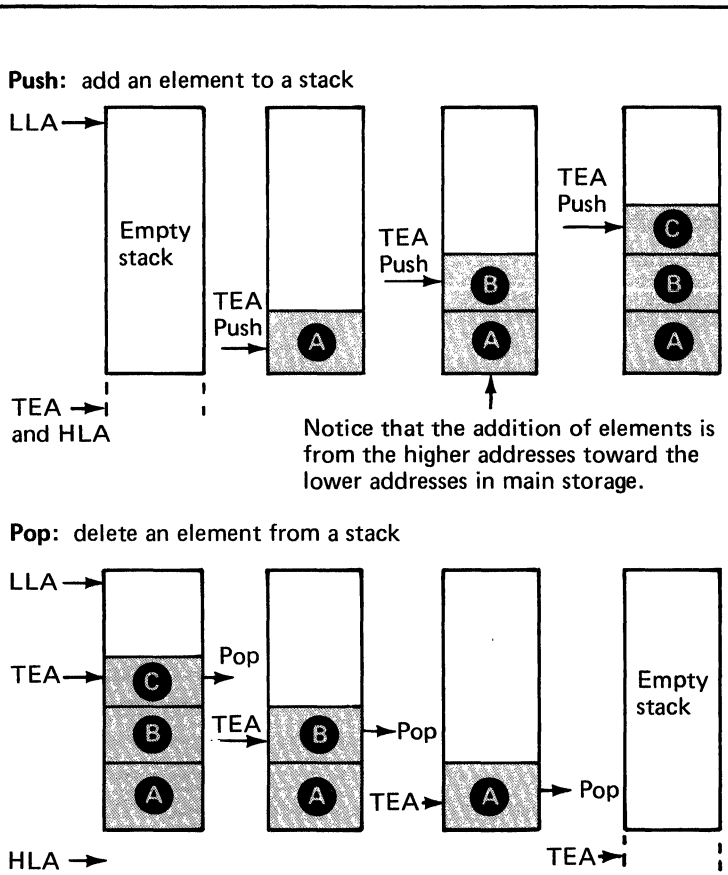


Figure 26. The relationship of the stack control block to the data stack



The Series/1 provides instructions for adding and deleting one-, two-, and four-byte size elements.

The processor handles all pointer updating and error detection.

When allocating main storage for the stack, the user must insure that the area allocated is a multiple of the size element being utilized.

Elements are added to the stack at the same end from which they are deleted (last-in-first-out—LIFO).

Figure 27. Adding and deleting elements from a stack

Push Operation. When a new element is pushed into a stack, the address value in the top element address is decremented by the length of the element (one, two, or four bytes) and compared against the low limit address. If the top element address is less than the low limit address, a stack overflow exists. An interrupt occurs with the stack exception bit set in the program status word. This interrupt is an example of an internal or class interrupt discussed earlier in this chapter. The top element address is unchanged. If the stack does not overflow, the system updates the top element address and moves the new element to the top location defined by the top element address.

Pop Operation. When an element is popped from a stack, the top element address is compared against the high limit address. If it is equal to or greater than the high limit address, an underflow condition exists. An interrupt occurs with the stack exception bit set in the program status word. If the stack does not underflow, the system moves the stack element—defined by the top element address—to the specified register, and increments the top element address by the length of the element.

Note: It is possible to pop data from beyond a stack boundary if:

1. The top element address is less than the high limit address, *and*
2. The operand size is greater than the high limit address minus the top element address

Data Stacking Example—Allocating Fixed Storage Areas

Many programs require temporary main storage work areas. Users find it very economical to be able to assign, dynamically, such work-area storage to a program only when that storage is needed. Conversely, when work-area storage is no longer needed by a program, it is economical to free that resource so that other programs can use it. The stacking mechanism can assist the user in programming the dynamic storage management function.

The following paragraphs describe how a user could allocate storage areas using the stacking mechanism (Figure 28).

A stack is initialized with an address that points to a fixed area of storage. Each element in the stack represents the starting address of a block of storage consisting of 512 bytes (e.g., addresses 0200 through 03FF). As storage is needed, the system pops the starting address for a block of storage from the stack. When the system no longer needs the block of storage, it pushes the starting address back into the stack.

The stack control block, and the stack and storage areas appear initially as shown in Figure 28 (1 of 4).

Notice that each stack element is one word long; addresses of the storage area are synonymous with the stack elements; the the top element address (TEA) points to the lowest location of the last element because the initialized stack is full. Contrast this with an empty stack (Figure 27) in which the top element address points to the same location as the high limit address.

Assume that program A requires a block of storage. Program A (or a storage management function at the request of program A) issues a Pop Word instruction against the stack control block. The system updates the top element address as shown in Figure 28 (2 of 4).

The system places the popped-word element in the register specified by the Pop Word instruction executed by program A. This is the address of the 512-byte storage area beginning at address 0200.

Assume that program B (operating on a different hardware level than program A) also requires a storage area as shown in Figure 28 (3 of 4). It, too, executes a Pop Word instruction against the stack. The next stack element is moved to the register specified and points to the next available storage area; then, the top element address is updated.

Before any further requests occur, program A terminates its need for a work area. Program A then issues a Push Word instruction against the stack and returns the address of the area it occupied so other programs can now use that same area as shown in Figure 28 (4 of 4).

Problem: Given a supply of buffer or storage areas of equal length—provide them, on a temporary basis, to tasks executing at different priority levels.

Solution: Use the stack as the list of available areas. Use the stack manipulation instructions to insure that interrupts of different priority cannot interfere with one another by interrupting during a crucial time—for instance, when storage areas are being allocated.

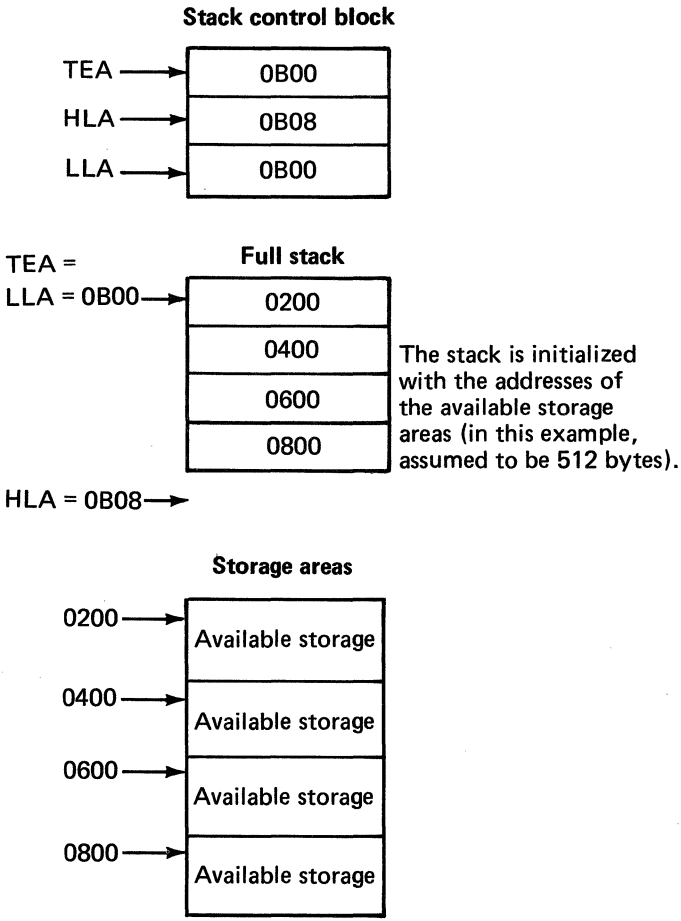


Figure 28. Example of stack usage: allocation of storage areas to concurrent programs (1 of 4)

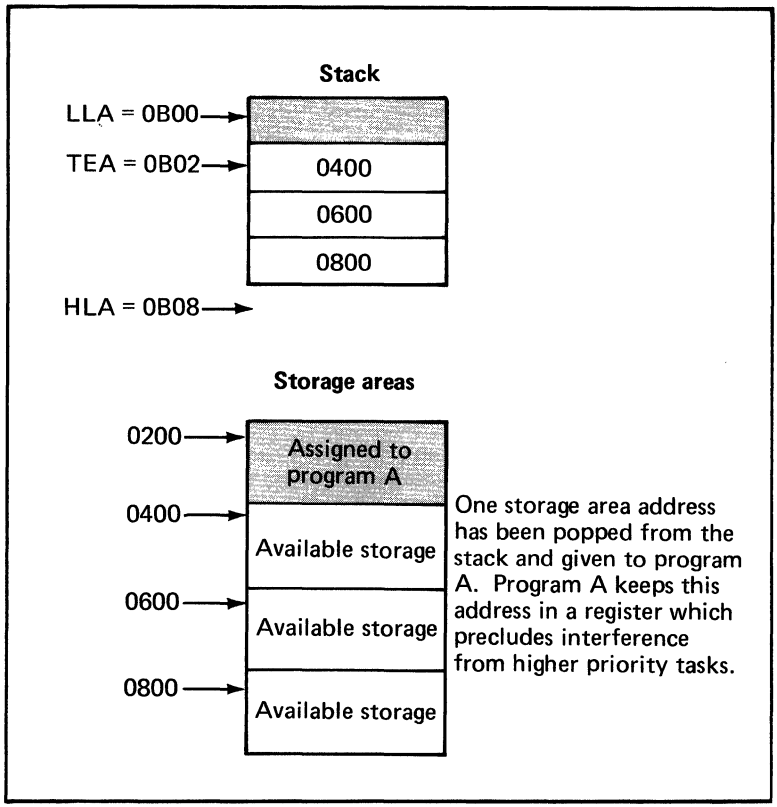


Figure 28. Example of stack usage: allocation of storage areas to concurrent programs (2 of 4)

A similar operation will be performed by program B when it releases its storage to the stack, popping address 0400 into location 0B00. While the addresses are obviously shuffled in the stack—the values differ from those initially established—no operational problems occur. This is so because each program requires only that an area of storage be assigned—it is not important where that area is located.

Linkage Stacking Description

A word-stack mechanism can be used for subroutine linkage. This mechanism saves and restores registers and allocates dynamic work areas (Figure 29).

A higher-priority program—B—interrupts and requests a storage area. A second address is popped from the stack and assigned to program B. Note that an interrupt cannot occur in the middle of a stack operation; consequently, one stack operation is always complete before another begins.

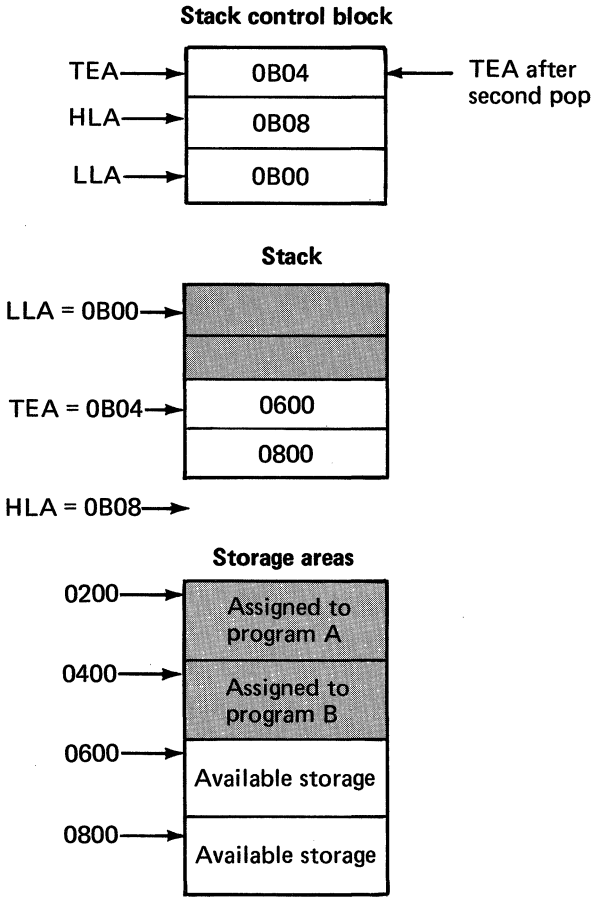
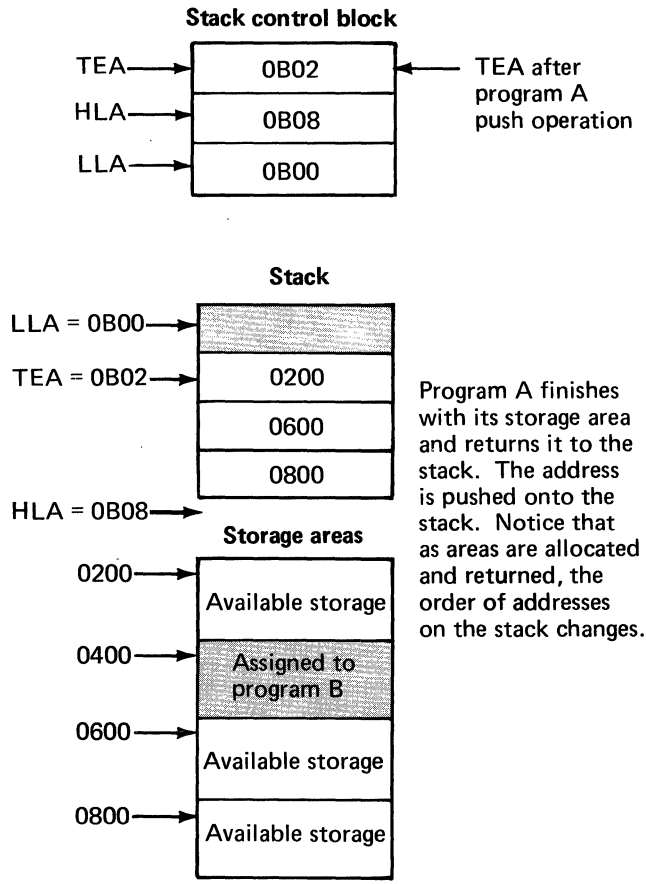
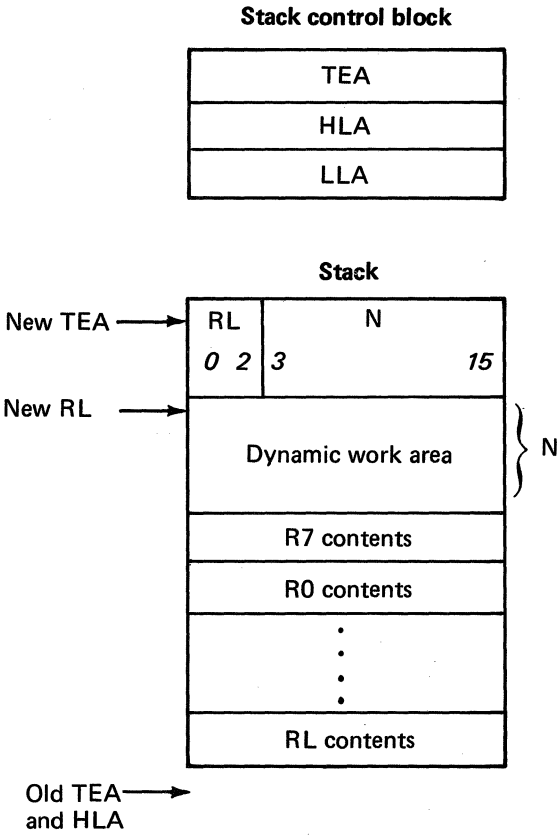


Figure 28. Example of stack usage: allocation of storage areas to concurrent programs (3 of 4)



Any program requesting storage may discover that no storage is available. An interrupt would then signal an empty stack of available addresses. The program must then decide whether to abort or wait for the storage to become available.

Figure 28. Example of stack usage: allocation of storage areas to concurrent programs (4 of 4)



Special instructions, Store Multiple and Load Multiple, reference a stack control block and specify:

- The number of registers to be saved
- The number of registers to be restored
- An arbitrary-sized area to be stacked (N)

This figure shows the use of special instructions to push and pop arbitrary-sized elements on a stack, and to facilitate reentrant programming.

Figure 29. Example of hardware and software integrated design

The Store Multiple (STM) instruction specifies:

- Stack control block address
- Limit register (RL) number
- Number (N) or words to allocate for work areas

When the STM instruction is executed, the requested block size, in words, is the sum of:

- The allocated value (N), plus
- The number of registers saved, plus
- One control word

The block size (converted to bytes) is used to decrement the TEA before making an overflow check. If no overflow occurs, the operation proceeds. The link register (R7) and register 0 through the specific limit register (RL) are saved, sequentially, in the stack. If register 7 is specified as the limit register, only register 7 is stored in the stack. The dynamic work space is allocated, and a pointer to the work area is returned in register RL. If no work area is specified, the returned pointer contains the location of R7 in the stack. The values of RL and N are also saved as an entry in the stack. The TEA is updated to point to the new, top-of-stack location.

When a Load Multiple and Branch (LMB) instruction is executed, the values of RL and N are retrieved from the stack and the system makes an underflow check. The value of RL controls the reloading of the registers; the values of RL and N are used to restore the stack pointer (TEA) to its former status. The contents of register 7 are then loaded into the instruction address register, returning program control to the calling routine.

Linkage Stacking Example—Reentrant Subroutine

Programs that operate on different interrupt levels may use the same subroutine. Instead of providing copies of the subroutine (one copy for each program that needs it), the subroutine can be made reentrant. That is, only one copy of the subroutine is provided and the single copy is used by

all requesting programs. Two items must be considered in the reentrant subroutine code:

1. Saving the register contents of each calling program. The subroutine is then free to use the same registers, restoring their contents to the calling-program's values just before the subroutine returns to the calling program itself.
2. Preserving the applicable variable data (generated by the subroutine) that is related to each call of the subroutine. This is done because data associated with one call must not be disturbed when subroutine execution is restarted due to another call from a higher priority program.

By using the STM and LMB instructions, the stacking mechanism handles items one and two, above. As an example, the operation could proceed as follows (Figure 30):

1. Program A calls the subroutine by means of a Branch and Link instruction (return address is in R7)
2. The subroutine, in this example, uses registers R3 and R4 during its execution. The subroutine receives (from program A) a parameter list address in R0, and the address of the stack control block in R1. Also, the subroutine executes, upon entry, the following store multiple instruction:

   ```
   SUBRT STM 4,(1),20
   ```

   After execution of the STM, the stack appears as shown in Figure 30. The last word contains a value that specifies the last register stored (R4 in this example) and the size of the dynamic work area (in words). During the STM operation, R4 (the last register stored in the stack) is automatically loaded with the address of the work area to be used by the subroutine to hold its work data.
3. When subroutine processing for this call is completed, the subroutine executes a single, Load Multiple and Branch instruction in order to reload the registers and return (via R7) to the calling program.

   If a second call to the subroutine has occurred prior to execution of the LMB, action similar to that just stated would occur again. However, another stack area would be

used. To complete processing for the first call, a return to the interrupted subroutine would occur when:

a. Subroutine execution is completed for the second call, and
b. All higher priority, interrupt-level processing is completed

In this way, multiple calls to a single subroutine are processed without interfering with the integrity of data associated with any other call to the subroutine.

Efficient use of the stacking mechanism depends on processor instructions for adding and deleting information. The stack, however, is a finite resource; consequently, exceptional conditions may occur which must be detected. These include overflowing the allotted stack area or removing more elements than are on the stack. The Series/1 instructions used to push and pop bytes, words, and doublewords, and the instruction used to store a group of registers and allocate an arbitrarily-sized work area on the stack (Load and Store Multiple instructions) contain hardware facilities to detect exception conditions and cause class interrupts. It is not necessary for user programs to test repetitively to determine if the stack has enough room. Rather, when the exception occurs, an interrupt response task can respond and do the appropriate error recovery pertinent to the particular use of the stack.

The ability to test and detect error conditions like these is easy and inexpensive in a microprogrammed processor. IBM has carefully designed the instruction set of the Series/1 to take advantage of this capability to make system software and application software as reliable as possible.

Interrupt Masking Facilities and the Interrupt Response Algorithm

It is usually, but not always, advantageous to respond quickly to asynchronous external events. Often application tasks or operating systems must update shared data items or manipulate other shared resources. If interrupting the operation might result in erroneous information being stored or

Problem: Provide an efficient mechanism for: 1) saving the registers of a program calling a subroutine, and 2) allocating to the subroutine a work area for its temporary use. The latter permits tasks which interrupt one another to share subroutines.

Solution: Use a stack—with special instructions that move registers to the stack—and restore register values from the stack. The same instructions must permit allocation of an arbitrary work area on the stack. Use the hardware-designed special instructions of the Series/1 to solve this software problem.

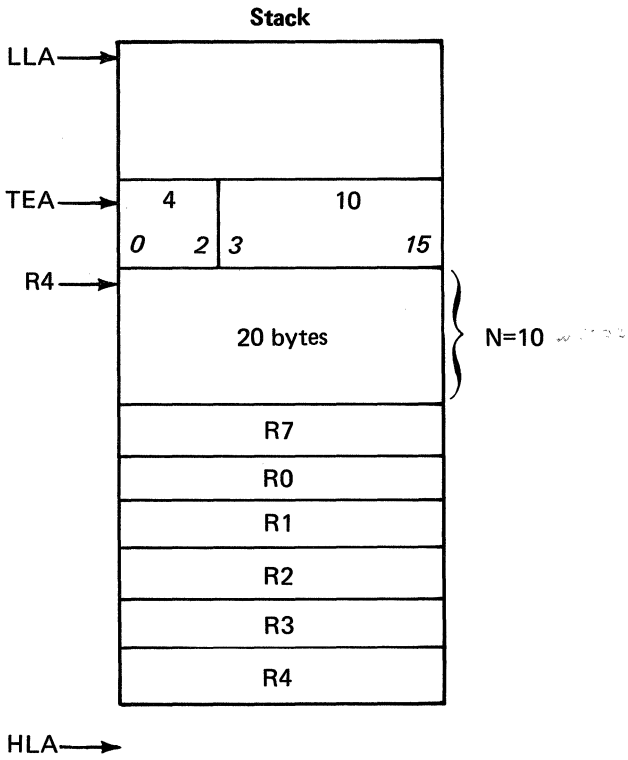


Figure 30. Example of stack usage: subroutine linkage and allocation of a work area (1 of 2)

In this example, ten words have been specified for the work area. Not all eight registers need be saved in this example. Registers 0 through 4 have been specified. Register 7 is automatically saved because it is used during the linkage process. Arguments of the subroutine are usually transmitted through the registers and, hence, are available to the subroutine on the stack.

Figure 30. Example of stack usage: subroutine linkage and allocation of a work area (2 of 2)

used by another task, the operation is termed critical. Such interrupts must be prevented. Usually, the user carefully designs such operations so they will execute quickly and infrequently to insure that the overall system response is not affected. Control over the interrupt mechanism is a useful tool for preventing interrupts that would adversely affect the system.

Three degrees of priority interrupt masking are provided for control of the interrupt processing:

1. Summary mask (bit 11 of the level status register)
2. Interrupt level mask register
3. Device mask

These registers, along with the conditions under which the processor responds to an interrupt, are shown in Figure 31.

Summary Mask

The summary mask supplies a masking facility for priority interrupts and certain class interrupts. The state of the summary mask (enabled and disabled) is controlled by bit 11 in the level status register (LSR) of the active priority level. When bit 11 is set to zero, the summary mask is disabled and prevents:

1. All priority interrupts regardless of priority level
2. Power/thermal and console class interrupts

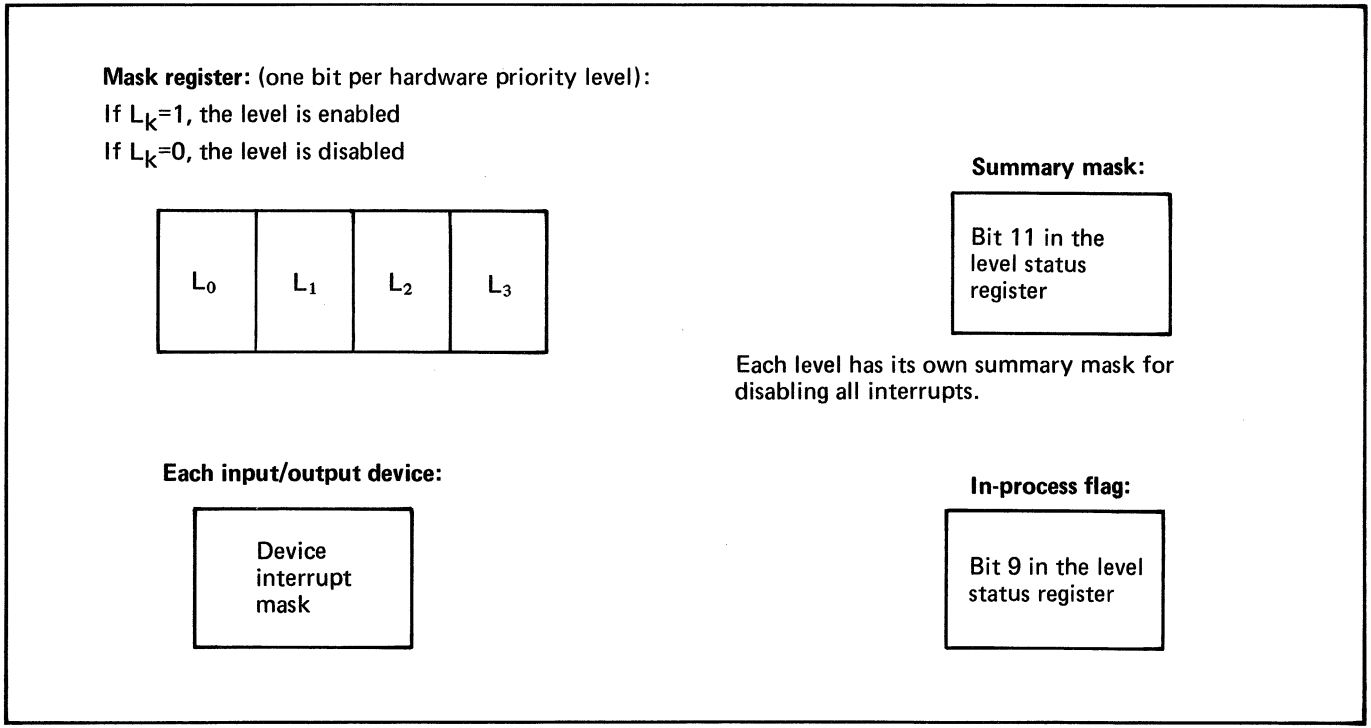


Figure 31. The priority interrupt algorithm (1 of 2)

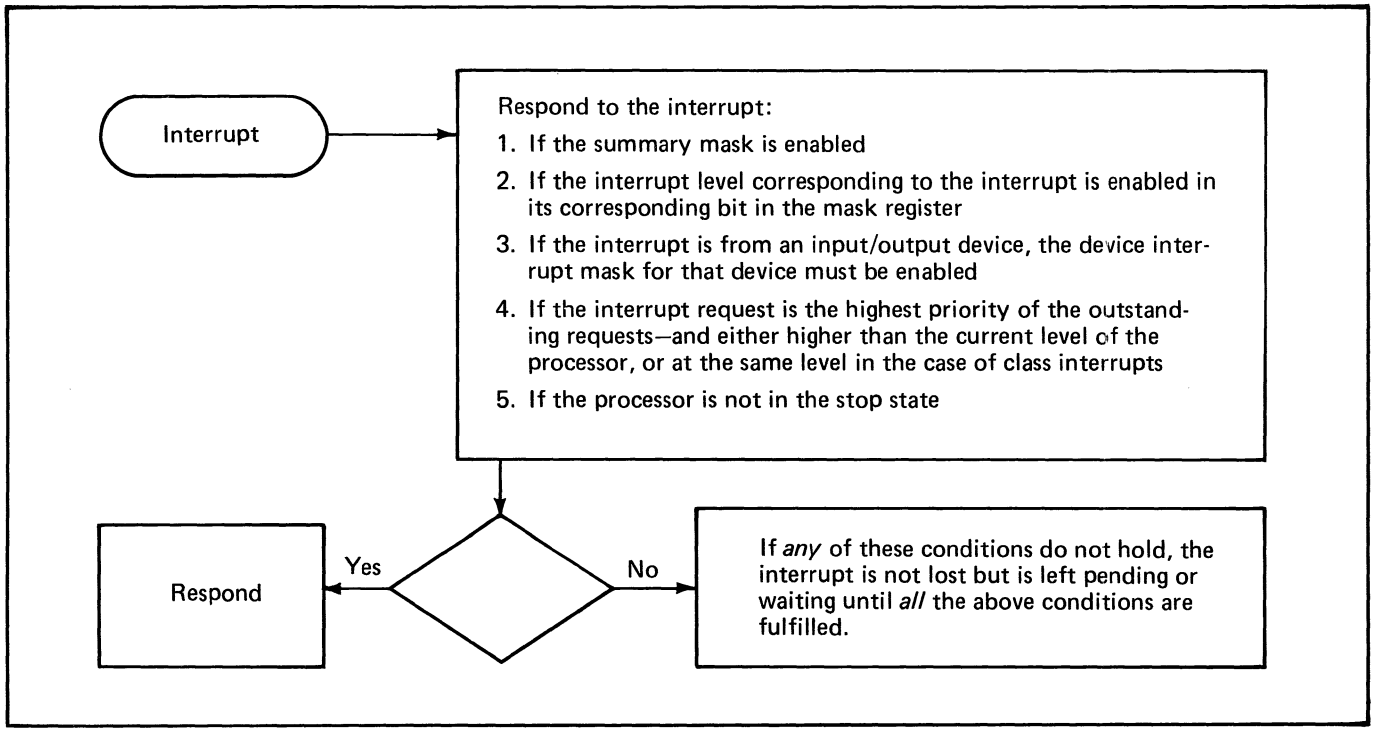


Figure 31. The priority interrupt algorithm (2 of 2)

All other class interrupts are enabled. When bit 11 is set to one, the mask is enabled and the interrupts are allowed. The summary mask is disabled and enabled as follows:

Disabled (Set to Zero)

- When a Supervisor Call (SVC) instruction is executed, the summary mask for the active level is disabled
- Execution of a Disable (DIS) instruction—with bit 15 of the instruction equal to one—causes the summary mask for the active level to be disabled
- All class interrupts disable the active-level summary mask
- The summary mask for a selected level is disabled by executing a Set Level Block (SELB) instruction with bit 11 of the LSR to be loaded, equal to zero
- The summary mask bits for priority levels 1–3 are set to zero by a system reset, power-on reset, or IPL

Enabled (Set to One)

- Execution of an Enable (EN) instruction—with bit 15 of the instruction equal to one—causes the active-level summary mask to be enabled
- The summary mask for a selected level is enabled by executing a Set Level Block (SELB) instruction with bit 11 of the LSR to be loaded, equal to one
- The level zero summary mask is enabled by a system reset, power-on reset, or IPL
- The summary mask for the interrupted-to level is enabled by a priority interrupt

Notice that the summary mask bit exists independently for each priority level. If the processor is in the wait state, the summary mask is enabled or disabled as defined by bit 11 in the LSR of the last active priority level.

Interrupt Level Mask Register

The interrupt level mask register is a 4-bit register used to control interrupts on specific priority levels. Each level

is controlled by a separate bit of the mask register as shown below:

Interrupt Level Mask Register				
Bit position	0	1	2	3
Priority level	0	1	2	3

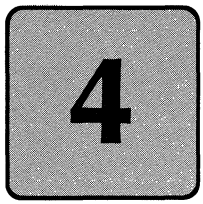
With a bit position set to one, the corresponding priority level is enabled and permits interrupts. With a bit position set to zero, the corresponding priority level is disabled. The system uses the Set Interrupt Mask Register (SEIMR) instruction to control bit settings in the interrupt level mask register. The Copy Interrupt Mask Register (CPIMR) instruction may be used to interrogate the register.

Note: All levels are enabled (set to one) by a system reset, power-on reset, or IPL.

Device Mask

Each interrupting device contains a one-bit mask called the device interrupt bit (I-bit). Interrupts by the device are permitted when its device mask is enabled (set to one). With the device-mask bit disabled (set to zero), the device cannot cause an interrupt. The device mask is controlled by a Prepare command in conjunction with an Operate I/O instruction.

The algorithm for responding to an interrupt, outlined in Figure 31, involves the priority of the currently executing level and the conditions of the various mask bits. The use of these interrupt masking functions is actually carried out with privileged instructions. In some dedicated types of applications, critical user-application tasks may execute in the privileged mode and directly manipulate interrupts. More often, the operating system maintains complete control over interrupts in order to successfully schedule and control concurrent sets of cooperating tasks—a condition typical of online, realtime applications.



Organization and Management of Main Storage

The objective of this chapter is to discuss the organization and management of main storage from the point of view of the overall architecture of the Series/1. Some of the capabilities discussed are hardware supported in some processors but not in others; some capabilities may not be supported in some software systems. Consequently, it is important to review the appropriate processor and software reference manuals when considering any single device in the Series/1 family.

The organization of main storage is central to the effective use of small computers. The hardware organization must both support a set of cooperating tasks and permit the use of efficient software for control of those tasks. To achieve compactness and speed, the Series/1 main storage itself is constructed using solid state FET (field effect transistor) technology. Supplied in up to 64K-byte increments, the maximum main storage supported is 64K bytes without hardware relocation translation and 256K bytes with the translator in the 4955 processor. Main storage speed is 300 nanoseconds with a restriction that 660 nanoseconds separate successive storage accesses. Each byte of main storage contains a parity bit for error detection. Specific details on speeds and parity bits vary from one processor model to another. The reader should consult the appropriate processor reference manuals for more specific information.

Main storage technology has been changing rapidly for the past few years and will probably continue to do so in the future. It is probable that main storage speeds, sizes, and reliability will continue to increase. As indicated earlier, IBM has deliberately designed the architecture of the Series/1 so that future technological improvements can more easily be incorporated in the system without obsoleting the design. The organization and use of Series/1 main storage, in particular, has been designed to make it compatible with these potential changes; hence, it is very important to consider how hardware and software cooperate in using storage.

Solid state main storage is volatile; that is, it loses its contents if power is lost. For those applications which cannot tolerate any loss of storage—or where it is difficult to checkpoint the application for restart from data kept in secondary storage—a battery backup unit is available. This unit is normally on stand-by and held at full charge. When a power failure is detected, this unit switches in to insure that the contents of main storage are not lost.

Chapter 3 discussed the processor and the input/output system access of main storage. It should be noted again, however, that in Series/1 processors with translation, all main storage addresses are actually 24-bits wide even though the largest main storage available is 256K bytes.

As described in Chapter 2, main storage is extensively self-checked. During processor power-on, the system checks the first 16K bytes of main storage for correct operation—general pattern checking also occurs. When using the Realtime Programming System, all installed storage is validated. Individual modules perform self-checking relevant to each module, and then check communications into main storage by uniting with and rereading a location. It is highly probable that a main storage failure will be discovered promptly.

User Concerns in Main Storage Organization

Since small computer applications are usually realized as a set of cooperating tasks, several aspects of main storage organization become important. They include:

Storage Addressing. Because multiple programs are co-resident in main storage, it is desirable to minimize the size of each. This minimization further reduces the time required to load the program from secondary storage and enhances response time of realtime applications. Although mainly related to instruction set sophistication, program size also relates to main storage word size and addressing. Specifically, effective use of registers for addressing purposes results in fewer full-word addresses and, consequently, more compact programs.

Address Space. Since small computer applications are usually structured as a set of relatively small, cooperating tasks, a 64K-byte address space (that is, the largest address that may be directly generated in an application program) is not often a limitation. More important, here, is the ability to use the full space for the task (and not share it with a large operating system, for example), and to access other tasks and data areas each of which may have its own address space.

Storage Protection. Reliability is the paramount concern in applications, but it is not realistic to expect all programs to be error free under all conditions. Instead, the user wants to be able to detect errors when they occur, trace them to their source, and respond in such a way that the application's objectives are still met. Detection of errors and error recovery is a function of both the interrupt system discussed in Chapter 3 and of the main storage organization.

Hardware Support for Reentrant Programs. Many software routines are shared among tasks. If the routines are reentrant, it is not necessary to delay the higher-priority task until the lower-priority task has completed its use of the shared routine. If such routines are to be efficient as reentrant routines, the system must provide storage addressing modes to permit instructions to reference different data areas at different times.

Efficient Intertask Communications' Capability. The word "cooperating" in the description of applications as a set of "cooperating tasks" cannot be overemphasized. The tasks share data, share routines, schedule one another, and perform many other functions. These tasks run concurrently, often in unpredictable sequences, as events occur. Main storage organization must permit data and routine sharing even though—at program preparation time—addresses of data and routines are not usually known. As in the case of reentrant routines, addressing modes must be present to do this effectively. Furthermore, the intertask communications must be consistent with the main storage protection mechanism so they do not reduce the reliability of the system.

Storage Management of Tasks. Either the Realtime Programming System or a special-purpose operating system created by the user manages the concurrent application tasks. In either case, the hardware organization of the main storage must facilitate: 1) getting tasks into and out of main storage with minimum overhead, and 2) switching execution rapidly from task to task. This is especially important in realtime applications and in processors with large main storage.

IBM designed the Series/1 main storage architecture with these user concerns in mind. Consider first a single task—involved in all of the aspects described above—when it addresses main storage.

Main Storage Addressing Modes

The user address space on the Series/1 is 64K bytes in length, corresponding to an address size of 16 bits; it is treated as an unsigned number. Storage references may be to bytes, words (pairs of bytes), doublewords, and quadruple words (as for example in double-precision, floating-point data). Bytes may be referenced at any address, but all words and multiples of words must start on word boundaries which are even-numbered bytes. Reference to a word is by the left-hand byte address. Hence, the left-hand byte of a word must reside at an even address and the right-hand byte of a word, at an odd address. This arrangement expedites the

accessing of words from main storage. Part of the instruction set's sophistication originates from its knowledge of the data type being addressed. For example, if it is known that a word is being addressed and hence its address is even, the low order bit—which is zero—need not be stored in the instruction. Of course, language translators for PL/I, FORTRAN, COBOL, and the assembler language take this into account at program preparation time.

The Series/1 provides a variety of addressing modes—all of which are useful in specific instances—for actually referring to main storage addresses. They may be divided into three broad categories:

1. Addressing modes which do not use registers
2. Register addressing modes
3. Based addressing modes

Direct and Indirect

In the first category of addressing modes, the address of the data item or main storage location is:

- 16-bits long
- Treated as an unsigned positive number
- Resides in a word in main storage as indicated in Figure 32

In that figure, *direct addressing* indicates a word which itself contains the desired address; *indirect addressing* indicates a word that contains a main storage address which, in turn, contains the desired address. Notice in the latter case that, before the system can access the data, an extra storage reference must occur to get the actual address. These modes of addressing can be used with many instructions but require a full storage word appended to the instruction to contain the address. This requirement lengthens the instruction from one word to two or, in the case of storage to storage instructions, to three words. In Figure 32, the examples show simple references to names which have been defined elsewhere in either assembler language or a compiler language. The asterisk symbol after a name means that the location named contains the address of the desired data rather than

the data itself. Direct addressing is useful in those situations where locations of routines and data are known at program preparation time.

Even in these cases, however, it is often more efficient to load the known addresses into registers and then use one of the register and based addressing modes to decrease program size. Indirect addressing can obviate the need to know addresses at program preparation time. As shown in the example in Figure 32, a user might refer to a routine whose address is not known by: 1) referring indirectly to a known location; 2) then, before the program begins execution, loading the address of the routine into that location. Indirect addressing is also used in the other two categories of addressing.

Register Modes

The second category of addressing modes is the set of register addressing modes shown in Figure 33. The first two modes illustrated there are similar to the direct and indirect modes discussed above because the system uses 16-bit addresses to address either the data or a main storage location which contains the address of the data. They differ because, in the second category, the addresses are in one of the eight general purpose registers rather than in a main storage word. This placement simplifies the instructions because they have to reference only one of the registers (two or three bits, depending upon the instruction) rather than a full word appended to the instruction. Furthermore, references to a data item cite a register containing that item's addresses; to move a data item, only that register's contents need be changed. This expedites referencing data separate from tasks, data in tables, and other data sources.

The register addressing modes contain one additional capability: namely, the *post incrementing* addressing mode. In this mode, denoted by a + sign after the register number, the system accesses the register to find the data address stored in that register—just as in register direct addressing; but the address in the register is then incremented by the length of the data item addressed. As a result, each reference

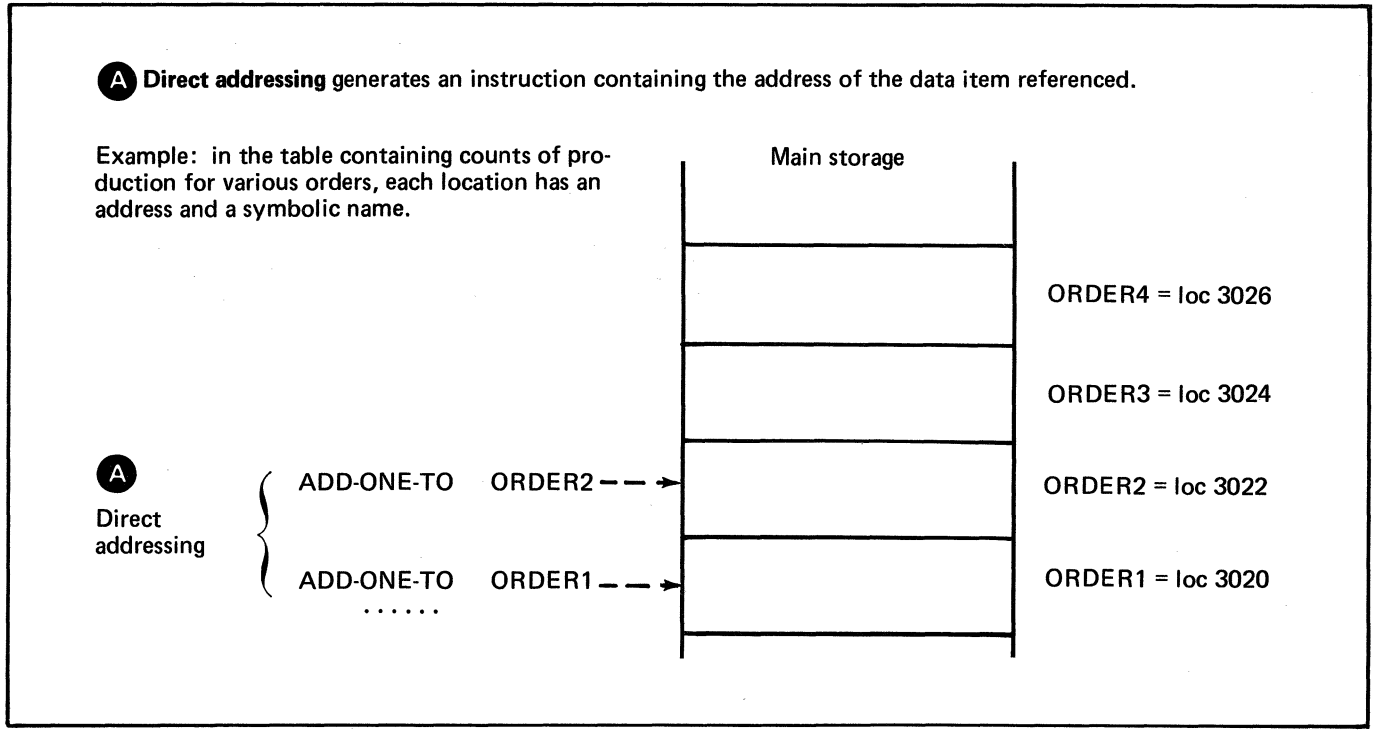
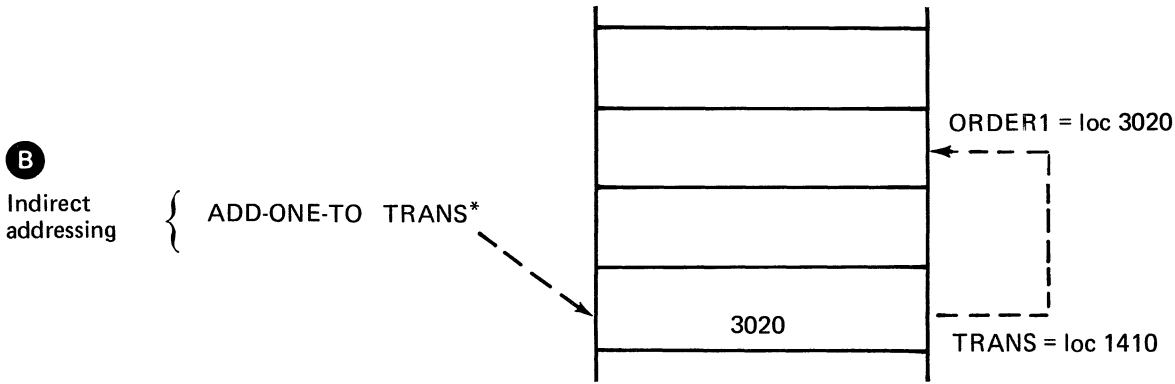


Figure 32. Storage addressing modes which do not use registers (1 of 3)

B **Indirect addressing** generates an instruction containing the address of a storage location; this storage location actually contains the address of the data item referenced.

A storage location with a symbolic name is used as a pointer to the order currently being referenced.



Indirect addressing allows programs to refer to different data items at different times by changing only the contents of TRANS rather than all of the addresses within the program.

Actual addresses—which appear as part of an instruction or as an indirect address—are a full 16-bits long and occupy a full word in storage.

Figure 32. Storage addressing modes which do not use registers (2 of 3)

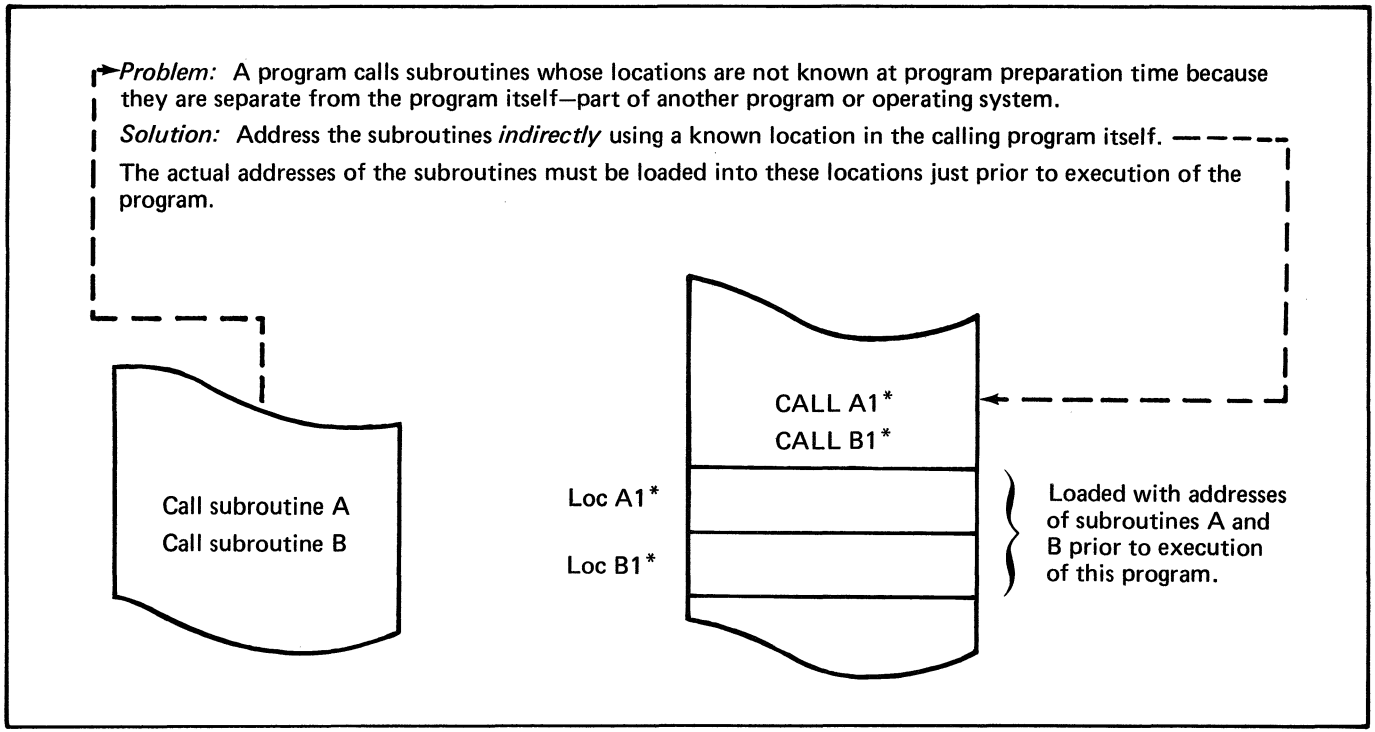


Figure 32. Storage addressing modes which do not use registers (3 of 3)

to a register with post incrementing mode changes the address. This procedure is very useful for sequencing through a table of items because it eliminates extra instructions that add constants to the register contents. The incrementing is totally automatic in the sense that the amount to be incremented (one byte, one word, two words) is determined by the instruction involved—because the data type referenced is implicit in the instruction. Register addressing modes are used extensively to minimize program size. Code generation in compilers is carefully designed to take this into consideration enabling application programs written in FORTRAN, COBOL, or PL/I to produce efficient object programs. An important part of assembler language programming is planning to permit use of these addressing modes rather than the longer, non-register modes.

Based Addressing

Based addressing, the third category of addressing modes, provides the real power for intertask communications, sharing data, and other functions required by a set of cooperating tasks. Figure 34 shows the three modes in this category. *Base relative* addressing uses one of the general registers as a base register; that is, a register which contains an address to be used in relative addressing. An item is referenced relative to that base address by providing its *displacement* from that address. The net address then consists of the base register number and the displacement as shown in Figure 34. This is the addressing mode used in most large computers including the IBM System/370. It permits the referencing of a number of data items via displacements; the displacements are usually small because programs are designed to be compact.

Figure 35 shows a data table containing several items of information. Base relative referencing of that data involves loading the selected base register with the starting address of the table (which need not be known at program preparation time). Each item is referenced by its known displacement from the beginning of the table. Notice that the table could be moved without changing the relative addresses of the data

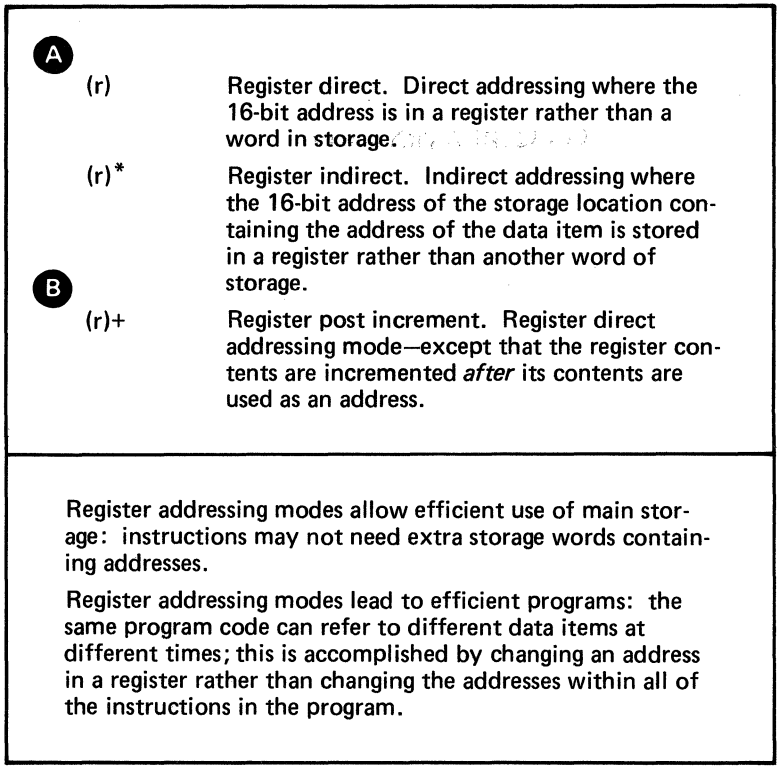


Figure 33. Storage addressing modes using registers for address storage (1 of 2)

within the table; to do so, the user has to change only the beginning address of the table in the base register. Certain instructions permit registers 1 through 7 to be base registers while other instructions restrict the choice to registers 1 through 3—whichever registers are appropriate for the instruction under consideration. Other instructions also limit the maximum displacement that a user may specify. This limitation is a value appropriate to the particular instruction. Maximum displacements range from a low of 31 bytes to a high of 32K bytes.

Indirect and Base Relative

Indirect addressing in the base relative mode is also permitted as shown in Figure 34. In this case, the user can choose when to perform the indirect part of the address

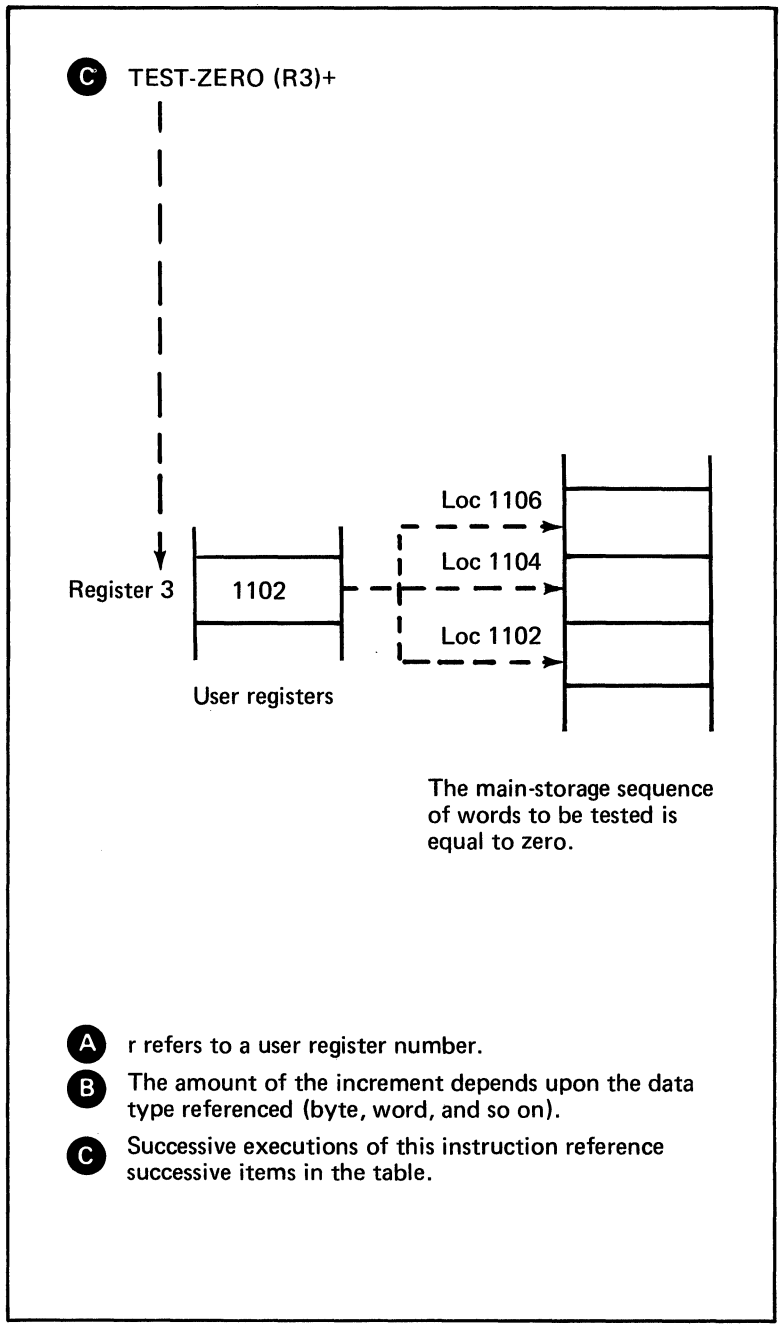


Figure 33. Storage addressing modes using registers for address storage (2 of 2)

Base relative addressing combined with indirect addressing

Pre-base relative indirect: add the displacement to the base register contents and use the result as an indirect address.

Post-base relative indirect: use the base register contents as the indirect address. Fetch the address in that location and then add the displacement to get the final address.

Pre- and post-base relative indirect: apply one displacement to the base register contents to get the indirect address. Add the second displacement to the address found in the indirect location to get the final address.

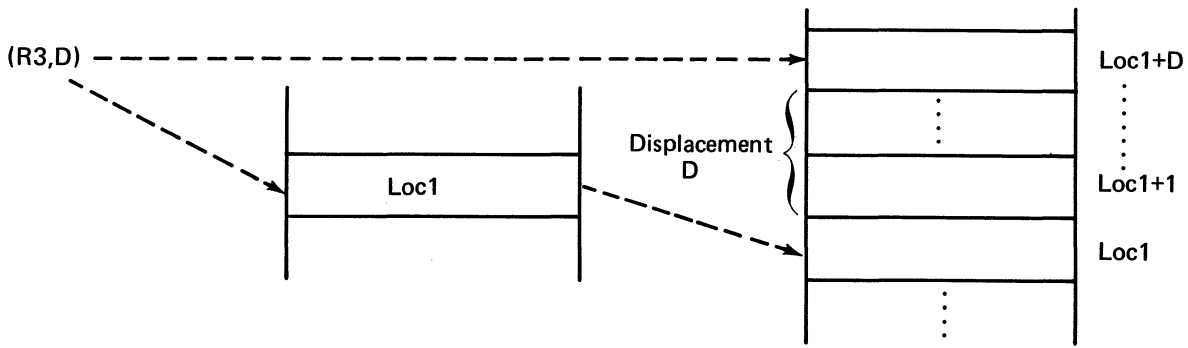


Figure 34. Base relative addressing and its variations (1 of 2)

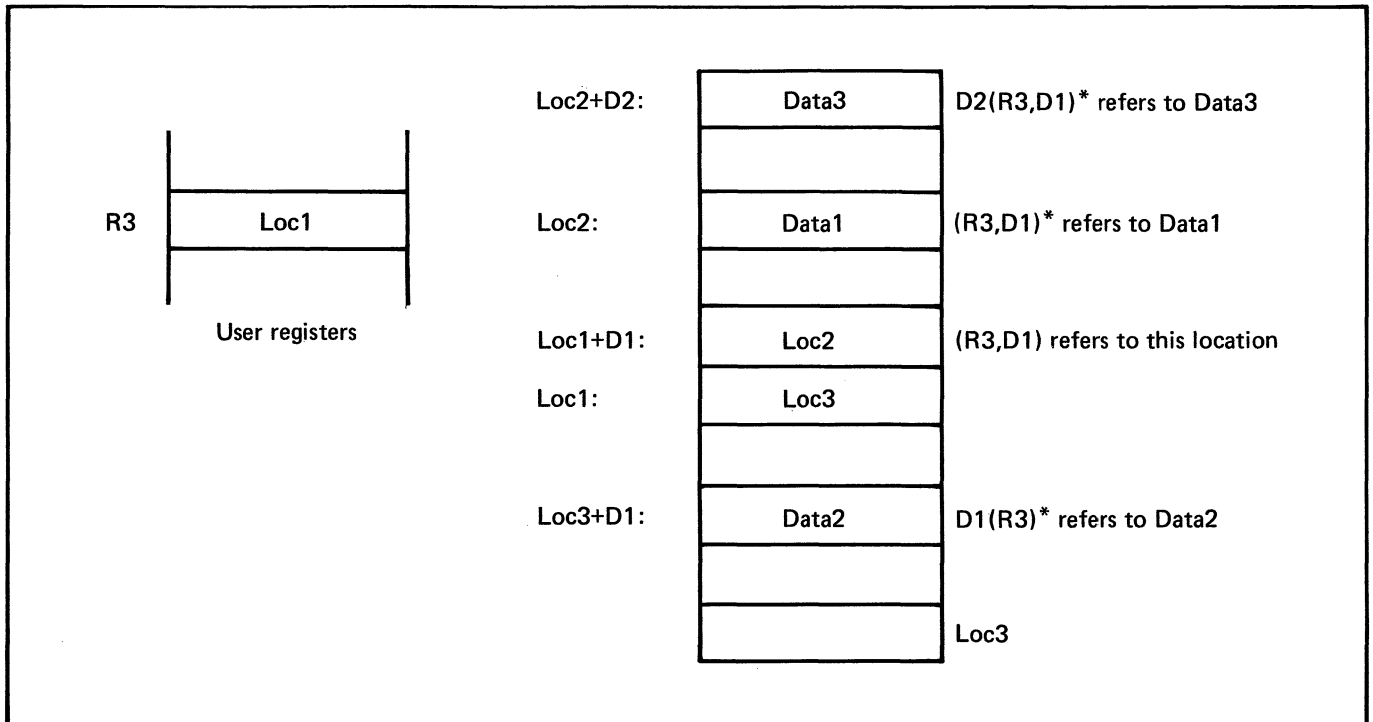
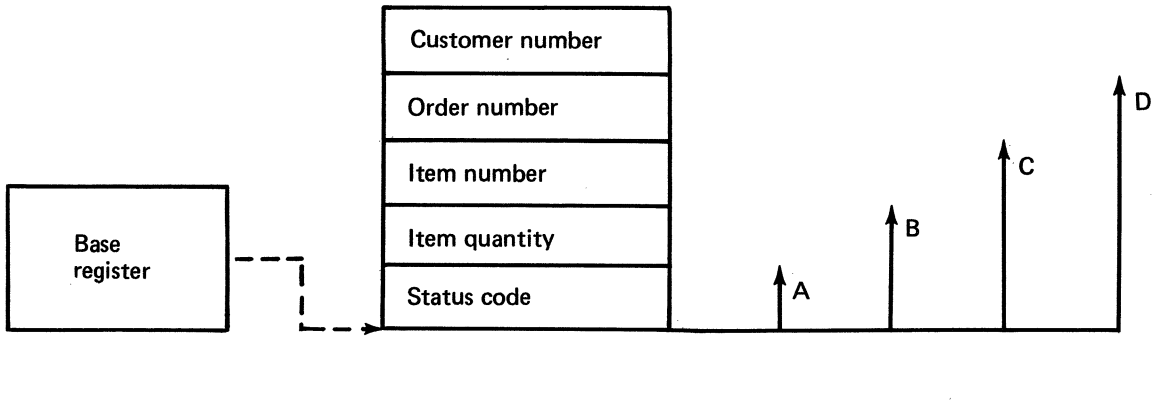


Figure 34. Base relative addressing and its variations (2 of 2)



Symbolic references to any data item in the data base will be transformed, where possible, to a base relative reference by program-preparation software. This minimizes instruction lengths and overall program length.

Reference to another customer's order would entail change to the register to point to the new data area, but would require no change to programs referencing items within the area.

Loading a base register with the address of the first item in the data base allows all other items to be referenced by their relative displacement.

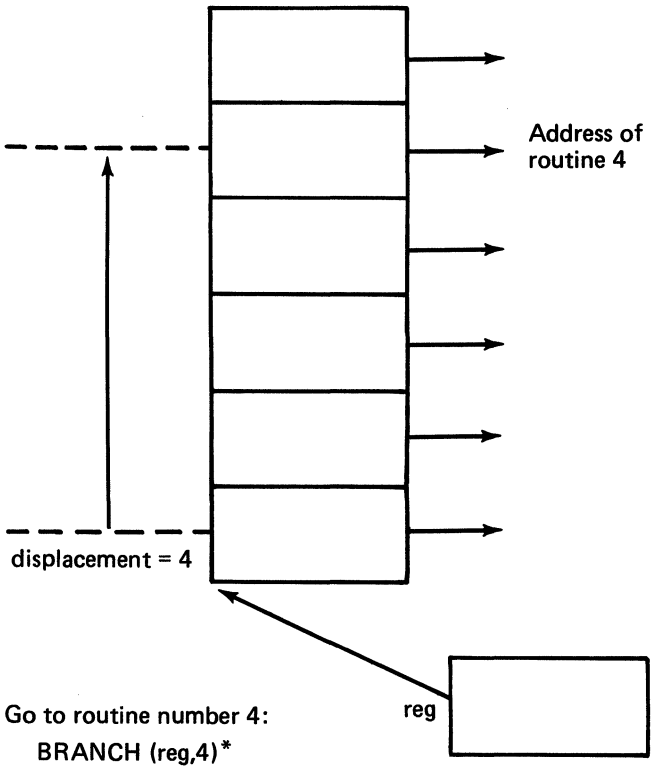
Figure 35. Base relative addressing of items within a contiguous base

cycle. As the figure indicates, in pre-base relative indirect addressing, the displacement is added to the indicated base register contents and that address is used as an indirect address. In post-base relative indirect addressing, the displacement is added after the register contents have been used as an indirect address. Both addressing modes are very useful as the examples in Figure 36 show. Both modes encourage the use of tables of addresses and tables of data items; these tables, in turn, simplify documentation, program updates, and online interaction among tasks.

Not only are pre- and post-base relative indirect addressing useful in themselves, but their combination can be very economical. The Series/1 provides that combination mode which is illustrated in Figure 34. Two, eight-bit displacements are permitted in this addressing mode; before actually accessing the data, the system adds one of them to the register contents before the indirect step and adds the other to the address resulting from the indirect step. Figure 37 shows how useful this can be when organizing data tables in a directory or hierarchical form. Clearly, the same effect could be obtained by loading addresses into registers and adding displacements, but those procedures would slow program execution and increase program size. A less obvious advantage of the combined addressing mode originates from the fact that all access to items is through displacements rather than addresses; consequently, tasks sharing data do not need to know the addresses of each data item—they need to know only the directory. Other tasks can modify and move data items provided the user updates directories properly. The IBM Series/1 Realtime Programming System makes extensive use of these facilities in its data management.

This variety of addressing modes means that there are often several choices for addressing a particular location or item. The Series/1 assembler can optimize this choice when it has the information available to do so. For example, the programmer can inform the assembler—symbolically—of the contents of the base registers. When a symbol is cited, the assembler attempts to reference it with a base relative address

References to a set of routines—each of which handles one elementary function—is common. A table of these routines' addresses can be referenced relative to the start of the table; it is followed by an indirect access.



Go to routine number 4:

BRANCH (reg,4)*

reg is the number of the base register.

Figure 36. Combined base relative and indirect addressing mode solutions to programming problems (1 of 2)

References to items within a data base are conveniently done relative to the beginning of the data base, as shown in Figure 35. If multiple, similar data bases are present—one for each order for example—it is convenient to: 1) use indirect addressing to point to the particular data base currently being processed; 2) then, reference items relative to the start of that data base.

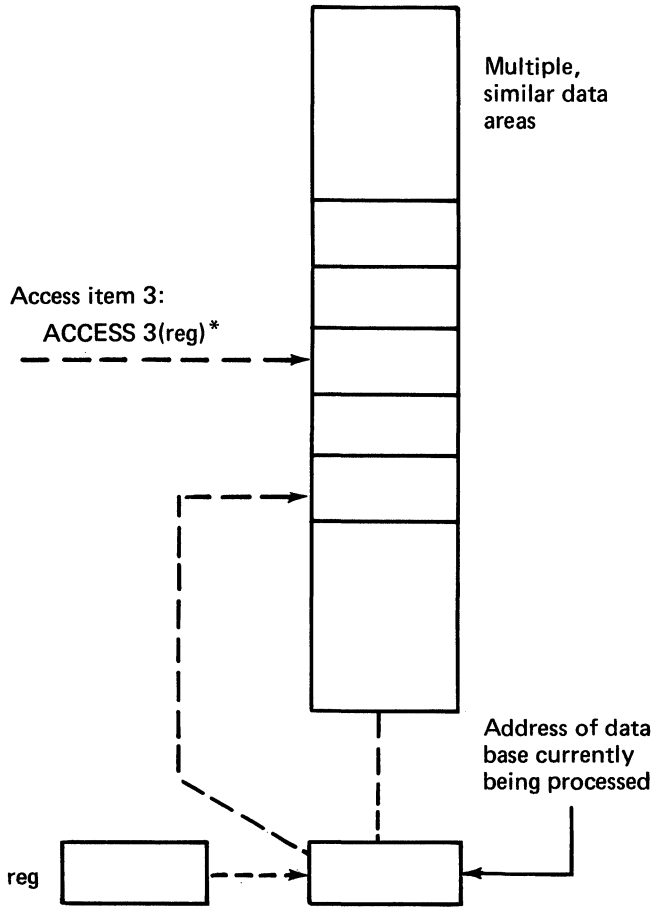


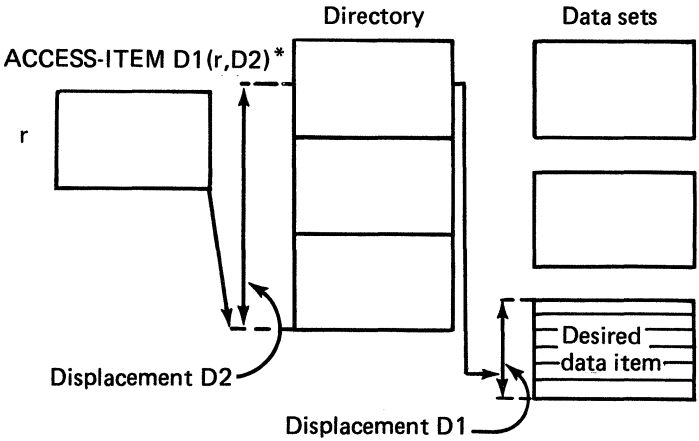
Figure 36. Combined base relative and indirect addressing mode solutions to programming problems (2 of 2)

Problem: Maintain multiple data sets—whose position can vary with time—so that users can efficiently access the data items within any one set.

Solution: Provide a directory in which the current address of each data set is maintained. Let this directory be referenced relative to its start (displacement).

Maintain the address of the directory in a register. Let users reference the directory using:

- The register as a base register
- The displacement within the directory to access the starting address of a specific data set
- Items within the data set, via a displacement from its beginning



Using combined addressing mode, all of the above processing may be done within one instruction; consequently, there is no problem of concurrency or need for programs to access these addresses and store them within the program itself—the system maintains control of the entire data base and can assure its integrity.

The illustrated combination of pre- and post-base relative indirect addressing permits easy implementation of efficient hierarchical files.

Figure 37. Combination of pre- and post-base relative indirect addressing

rather than a direct address. The assembler picks the shorter instruction.

Other addressing modes are used in specific instructions where appropriate but are not included in the above categories because of their restricted use. For example, Chapter 3 discussed the stack referencing instructions which perform complex operations on the addresses. Chapter 6 discusses special-instruction addressing modes together with the specific instruction involved in each.

Excluded Modes

It is also important to note that certain addressing modes are deliberately *not* present in the Series/1. For example, instruction address register relative addressing is not provided in the Series/1. This mode is often called relative addressing and, if consistently used, provides for position-independent code; that is, object code which can be executed at any arbitrary starting location in main storage. The Series/1 architecture obviates this addressing mode primarily because of its address translation capability which provides a more powerful position-independence than would be provided by relative addressing. The multiple levels of indirect addressing facility is not provided because it lacks general utility and it introduces problems of error detection and recovery.

When a user chooses an instruction set and storage addressing modes, some compromises in system design must follow. However, the *integrated* development of Series/1 hardware and software—both program-preparation software and operating-system software—has evolved an efficient set of main storage addressing modes consistent with the applications intended for the system. The precise descriptions of instruction formats, and the allocation of fields to those instructions, are covered in detail in the Series/1 reference manuals for the various processors. In Chapter 6, instruction formats are presented and their relationship to the addressing modes described here is discussed further.

Main Storage Protection

Reliability of the Series/1 is enhanced by an effective system for error detection and recovery. Main storage protection contributes to this system by enabling tasks:

- To protect critical areas from writing
- To protect critical areas from any access
- To connect the internal interrupt system to those tasks responsible for responding to violations

The precise means used for storage protection varies with the size of main storage. In particular, a small system with a main storage of 64K bytes or less ordinarily does not utilize hardware address translation as do systems with more than 64K bytes of main storage. Therefore, the small system needs its own protective mechanism because the built-in protection of the translator is not available. Protection for the smaller system is discussed in this section; protection for larger systems is discussed in the next section of this chapter.

Address Key Protection

In a 64K-byte (maximum) system, all tasks, data areas, and the operating system (the Realtime Programming System or a special-purpose operating system) can access the same 64K-byte address space; consequently, they must be protected in a way different from the way they would be in an environment in which they use different address spaces. For this purpose, the *address key* concept is used as shown in Figure 38. Main storage is shown divided into 2K-byte segments (there are a maximum of 32). Associated with each segment is an eight bit *storage key register* which controls access to that segment of storage. The system uses three of the eight bits as a key; that is, any integer between 0 and 7 is used to match a similar key in any task attempting to access the storage segment. One bit of the storage key register is the read-only flag which is set to the value 0 if the segment can perform both reading and writing, and to the value 1 if the segment performs reading only.

Storage Access Types

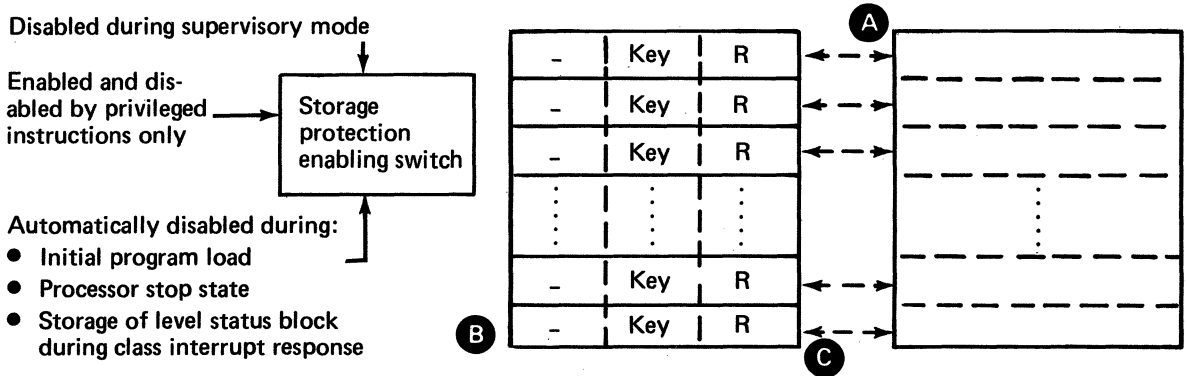
The processor uses three types of storage accesses: instruction fetching, source operand accessing, and destination operand accessing. As shown in Figure 38, three separate keys, called ISK (instruction space key), operand 1 key (OP1K), and operand 2 key (OP2K) are provided. When the processor is accessing main storage, the key appropriate to the item being accessed is termed the active address key and is used to determine whether or not the system permits access to the addressed segment of storage.

Storage protection is active when its enabling switch has been set by execution of a privileged instruction (Enable). Protection is automatically disabled:

- When the processor is in the supervisor mode
- During initial program loading
- While storing level status blocks in response to a class interrupt

Storage Access Checking

Provided storage protection is enabled, the microprogrammed hardware goes through the sequence of tests indicated conceptually in Figure 39. If the addressed segment key is 7, the user has indicated that no protection is desired for the segment and any task may access the area. Even if all accesses are permitted, the user can still protect against change by setting the read-only bit in the storage key register for that segment. For any key except 7, the system checks protection against the currently active address key. Protection, then, depends upon the items being accessed. If there is a key match, access is permitted and only the read/write option must still be determined. If there is no match, an error has occurred (an attempt to access illegally), and a class interrupt is responded to on the same priority level as the executing task. A level status block is automatically saved and the protect check bit in the processor status word is set. The interrupt response checks for the cause of the interrupt by examining the processor status word register,



A 64K main storage is divided into 32 segments; each segment has a 2K-byte length.

B Storage key registers are 8-bits wide. The key field is 3-bits wide. Any storage segment may be assigned any 3-bit key using privileged instructions.

The R-bit (read-only flag):

- If 1, writing is prohibited
- If 0, read or write access is permitted

C One storage key register exists for each 2K storage segment. Access to the storage key registers is limited to privileged instructions. Both the processor and its input/output system obey the same rules—except cycle steal input, which ignores the read-only access control.

Figure 38. Storage key protection of main storage (1 of 2)

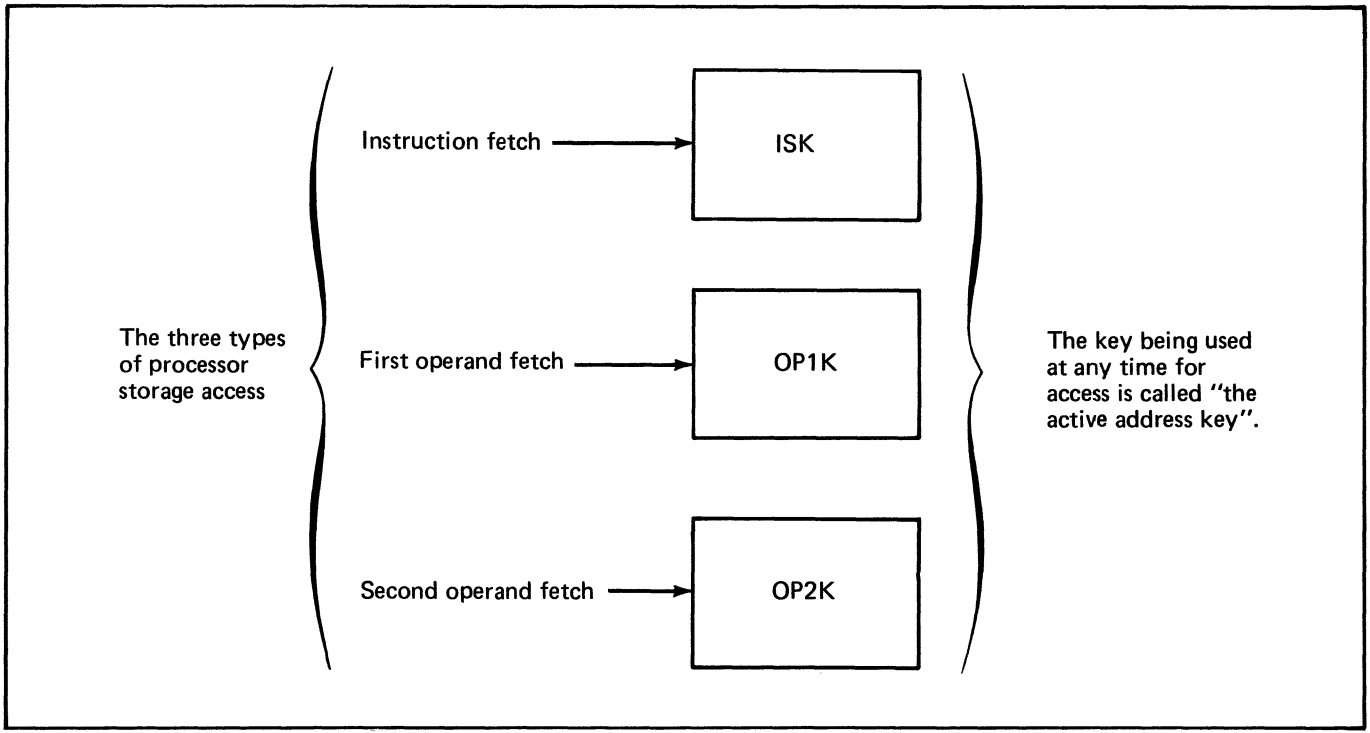


Figure 38. Storage key protection of main storage (2 of 2)

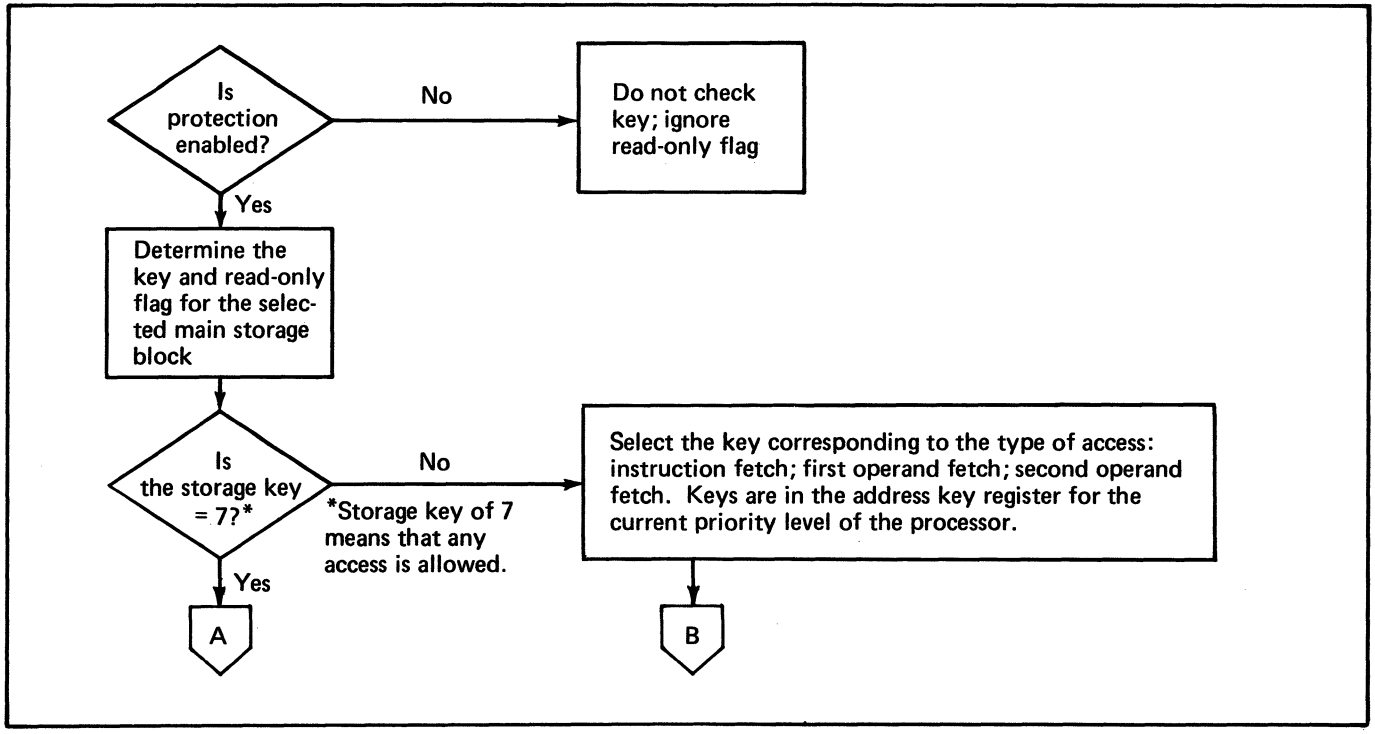


Figure 39. Operation of storage protection during an access (1 of 3)

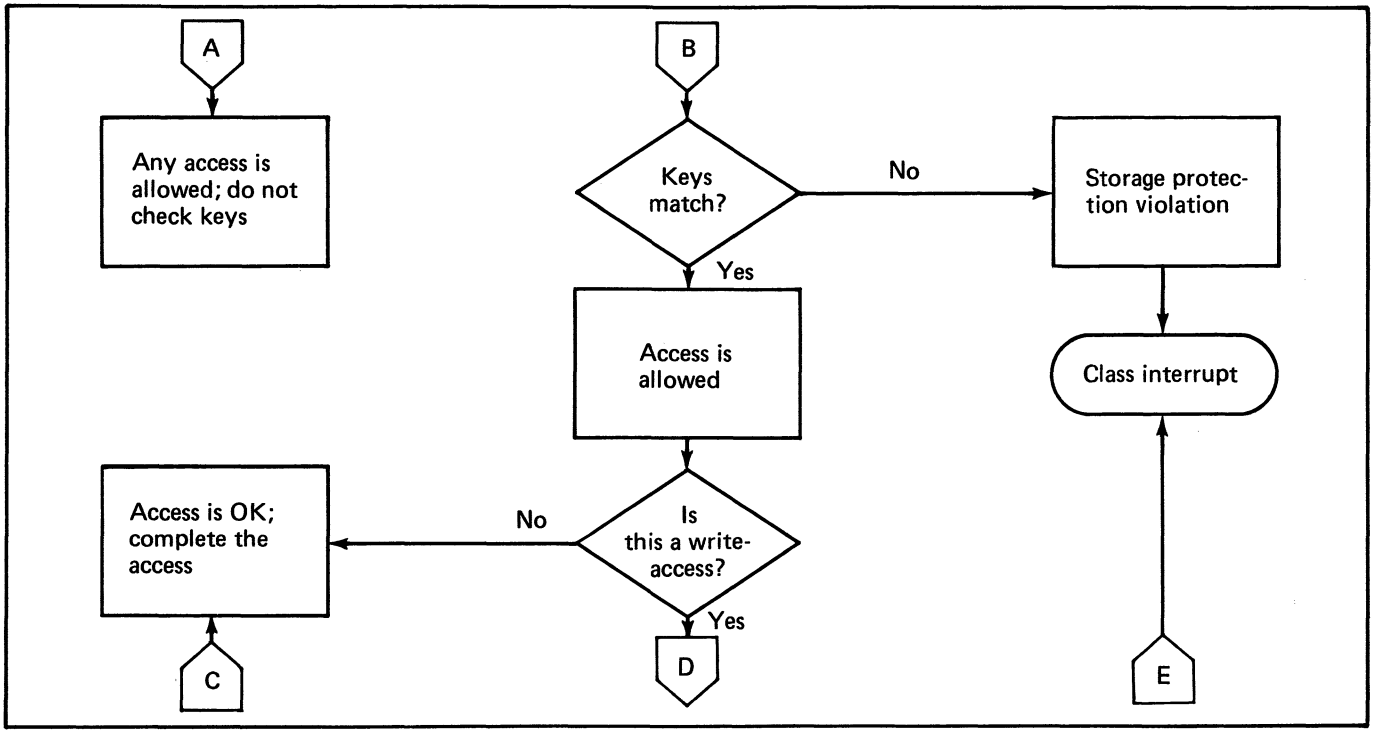


Figure 39. Operation of storage protection during an access (2 of 3)

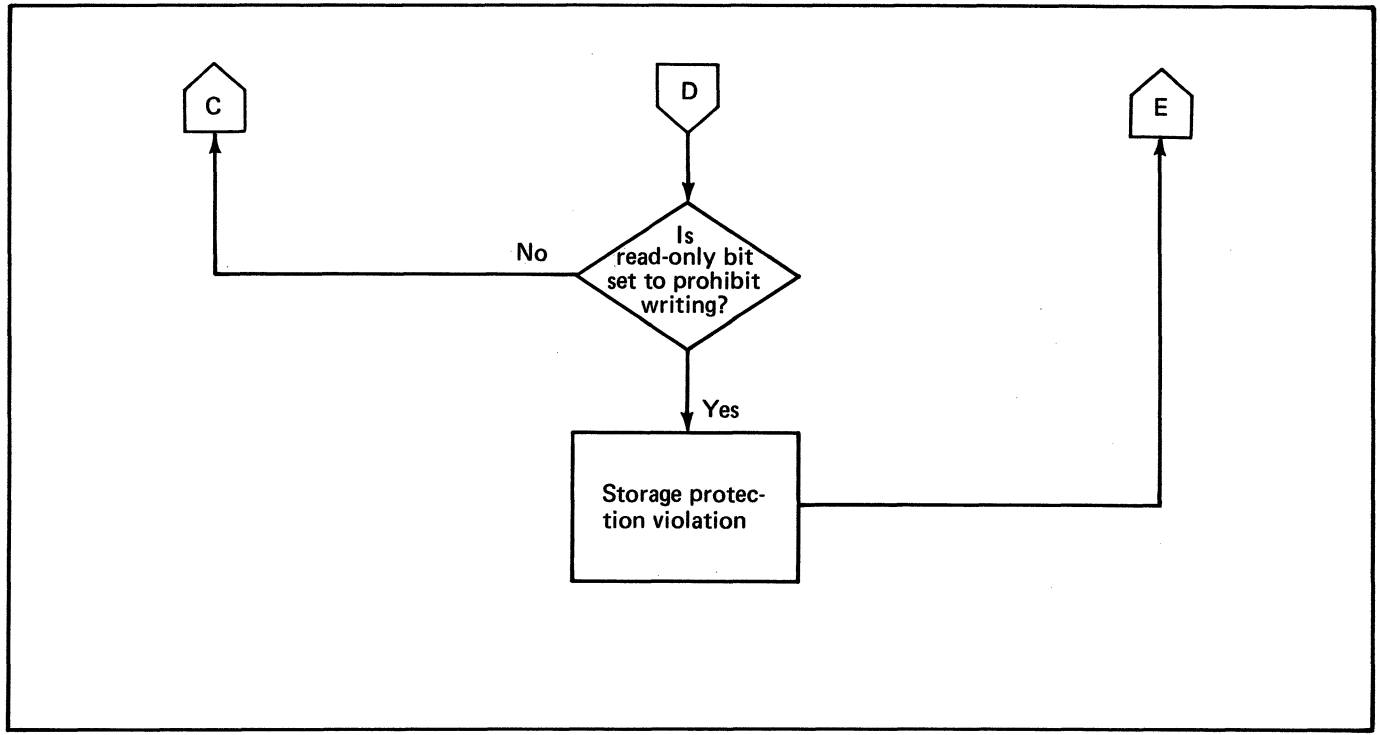


Figure 39. Operation of storage protection during an access (3 of 3)

and initiates the error recovery procedure appropriate to the application being performed.

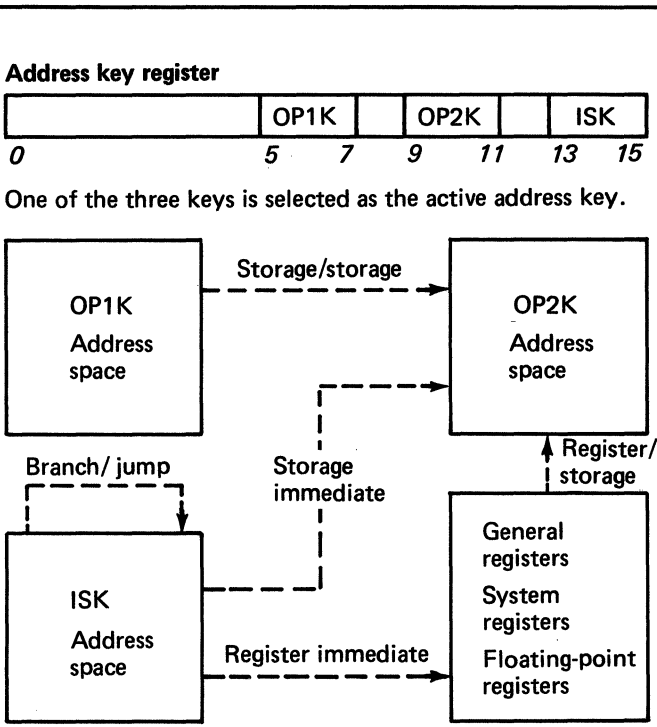
Figure 40 shows how the three address keys are used for various classes of operations. For example, a branch instruction fetches the program instruction and then causes the next program instruction to come from the location addressed. Both addresses are checked with the instruction space key. Similarly, the OP2K key is used alone if only one storage operand is referenced (the general registers have no storage protection key). Only if two storage operands are referenced in the same program instruction is the OP1K key used.

Multiple Task Protection

The method chosen for storage protection is determined by the kind of protection the user desires in an environment of cooperating tasks. Figure 41 shows three typical examples. First, a task which contains some read-only code; second, a data area which should not be changed; and third, a work area. In the first example, a key of 6 is assigned to the entire task and its data areas. The work area is set to read/write, and the read-only data area and code are set to read-only in the storage key registers. The address key register for this task would set all three keys to the same value: 6, in the example. Except for the read/write area, the task area is protected against other task access. The first example illustrated is an essentially self-contained task.

The second example in Figure 41 shows two tasks cooperating to update a common data area. Three different keys are assigned to each task and to the common data area. The address key registers for each task use identical OP1K and OP2K keys; as a result, each task can both read and write data in the common area. The instruction key is, of course, different for each task. Both tasks can then update and reference the common data area, but that area is protected against any task which does not have its key.

The third example in Figure 41 shows an important use of cooperating tasks where one task writes data into a separate data area and a second task reads that data. By



Each of the three address keys controls one type of storage access:

1. OP1K controls accessing of data from storage
2. OP2K controls data read into storage
3. ISK controls instruction fetches

Instructions which perform several such accesses use more than one key. A storage to storage transfer uses all three keys. A Branch instruction uses only the ISK key because the destination of a branch is an instruction location.

Key values may be the same or different. A storage key of 7 implies no storage protection (keys need not match). A key of 0 is, by hardware/software convention, considered special; it is used during input/output and by the operating system.

Figure 40. Use of the three storage protection keys by various classes of operations

combining read/write control and address keys, the user maintains control over the access of each task. Notice that the address key registers of each task use three different keys because the code is separate from the data areas.

In a custom-designed operating system, programmers can, at will, lay out their storage protection keys and use the system's supervisor and problem states to control the program's environment. Under the Realtime Programming System, the programmer maintains control at program preparation time but allows the operating system to control the accessing and changing of all keys and registers. Normally, the operating system is the only program executing privileged instructions (exceptions occur in user-supplied, privileged interrupt-response, and input/output programs). For the smaller system, the multiple-key approach gives the user genuine control over storage protection.

Main Storage Mapping Systems

For larger Series/1 systems, it is desirable for the system to be able to expand the size of main storage beyond 64K bytes. At the same time, it is not desirable to change the basic architecture of the system. To maintain continuity, IBM has added a mapped storage system which gives each task a 64K-byte address space—as in the smaller systems—but which also translates user-generated addresses to correspond to the physical storage address at execution time. This procedure initiates a fast responding, fast context-switching system, and also greatly improves storage protection.

Storage Segmentation

Mapping of main storage involves the division of storage into relatively short blocks (2K bytes in the Series/1), together with mapping or assigning each block to correspond with a set of addresses that the user generates. Figure 42 shows, conceptually, how this is done. The user task has an address space of 64K bytes and generates any address in the range 0 to 64K bytes. As shown in the figure, 11 of the

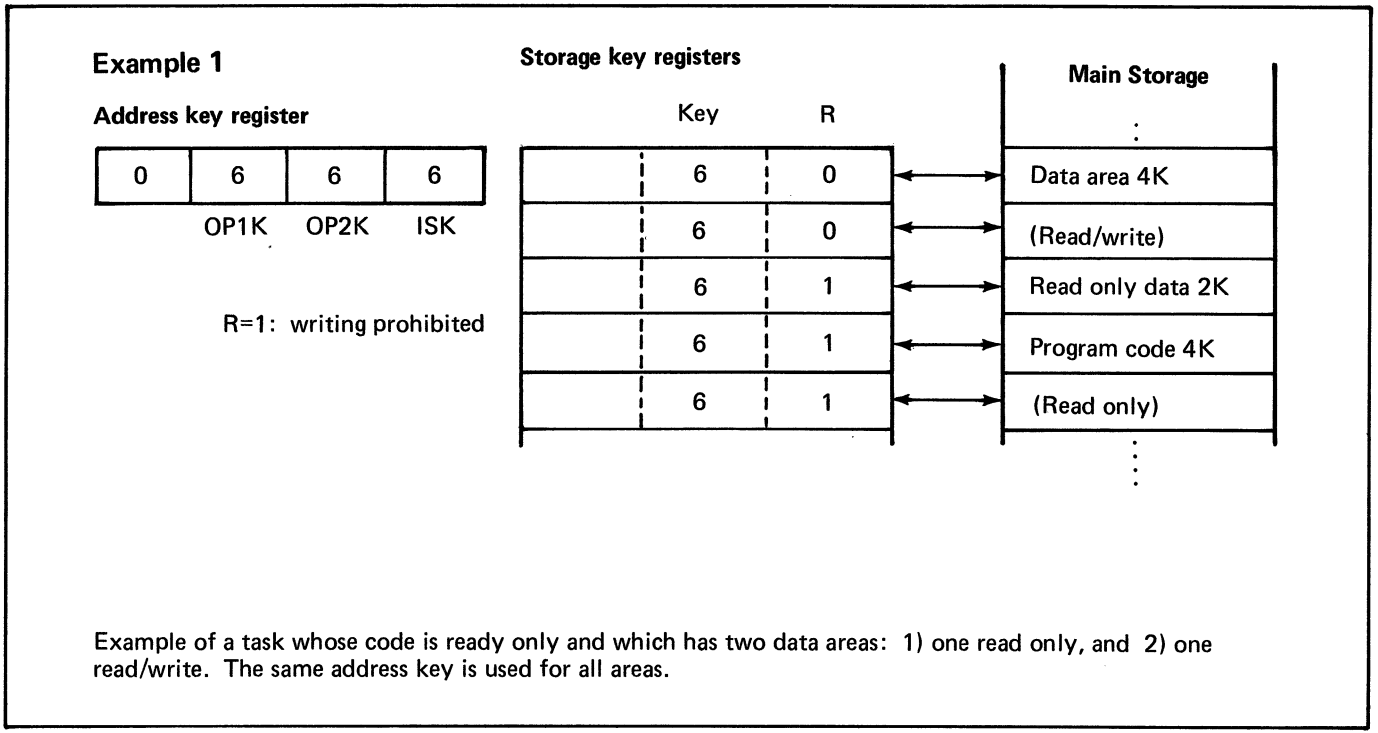


Figure 41. Three examples of address key storage protection (1 of 3)

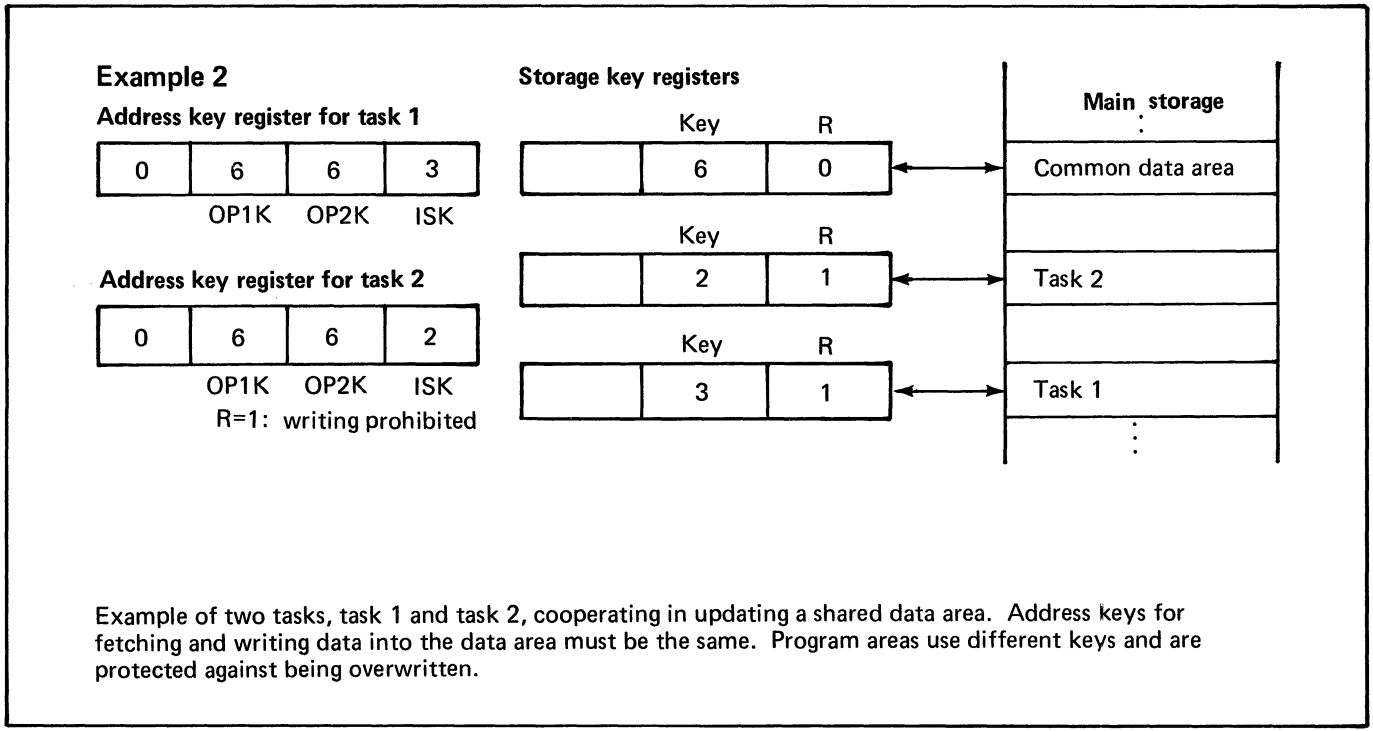


Figure 41. Three examples of address key storage protection (2 of 3)

Example 3

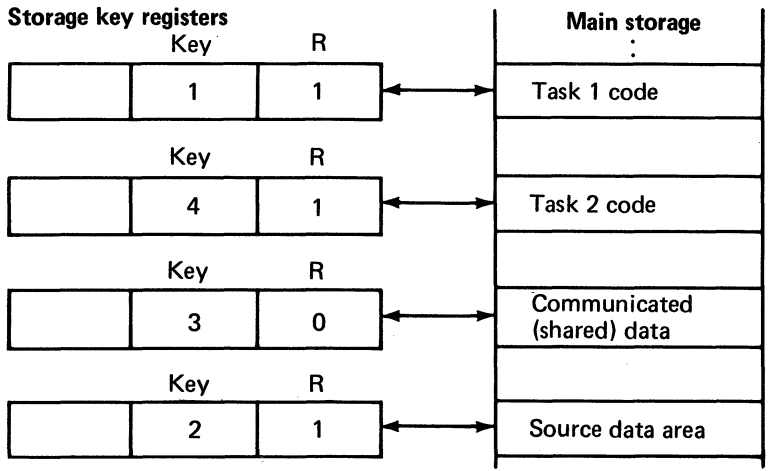
Address key register for task 1

EOS	OP1K	OP2K	ISK
0	2	3	1

If the equate operand spaces (EOS) flag is 1, OP1K becomes the same as OP2K.

Address key register for task 2

EOS	OP1K	OP2K	ISK
0	3	3	4



Example of two tasks communicating. Task 2 writes data from a protected area into an unprotected area for reading by task 1. (An example might be a data acquisition task using parameters stored in the area. Key = 2 inputs and processes data and stores it in the area. Key = 3 is used by an operator communications or control task.)

The equate operand spaces (EOS) flag is useful for tasks like the operating system which must move data from area to area.

Figure 41. Three examples of address key storage protection (3 of 3)

16 bits of the address are considered to be a displacement on a page (i.e., a 2K-byte block). The most significant 5 bits remaining are equivalent to the page number. The page-number portion of the user-generated address accesses one of the 32 segmentation registers to find the physical storage address of that page. The translator is responsible for:

- Accessing the segmentation registers
- Getting the physical page address
- Adding the displacement to obtain the actual physical address corresponding to the user-generated address

This hardware process executes rapidly and is now used widely in small computer technology.

In the Series/1, translation lengthens the main storage access cycle time from 660 nanoseconds to 880 nanoseconds. Note, particularly in Figure 42, that the physical segments making up what appears to the user to be a 64K-byte contiguous space may actually consist of *non-contiguous* blocks. Mapping frees the system from the necessity of fitting programs and data areas into contiguous space and the associated difficult main storage management that such an operation implies. Of even greater importance is a task's ability to address an area in main storage containing either shared data or shared routines just as though that area were physically contiguous with the task itself. This ability is especially important when multiple tasks cooperate in an application.

Mapping Multiple Tasks

The conceptual mapping of main storage, when multiple cooperating tasks reside there, is shown in Figure 43. The two tasks residing there may share routines and data areas by having the system invoke—in mapping registers belonging to each task—the physical segments containing those routines and data areas. Mapping provides an efficient mechanism for implementing the extensive, intertask communications required from responsive, small computer applications.

The main storage mapping of the IBM Series/1 is more complex than indicated above because the system must

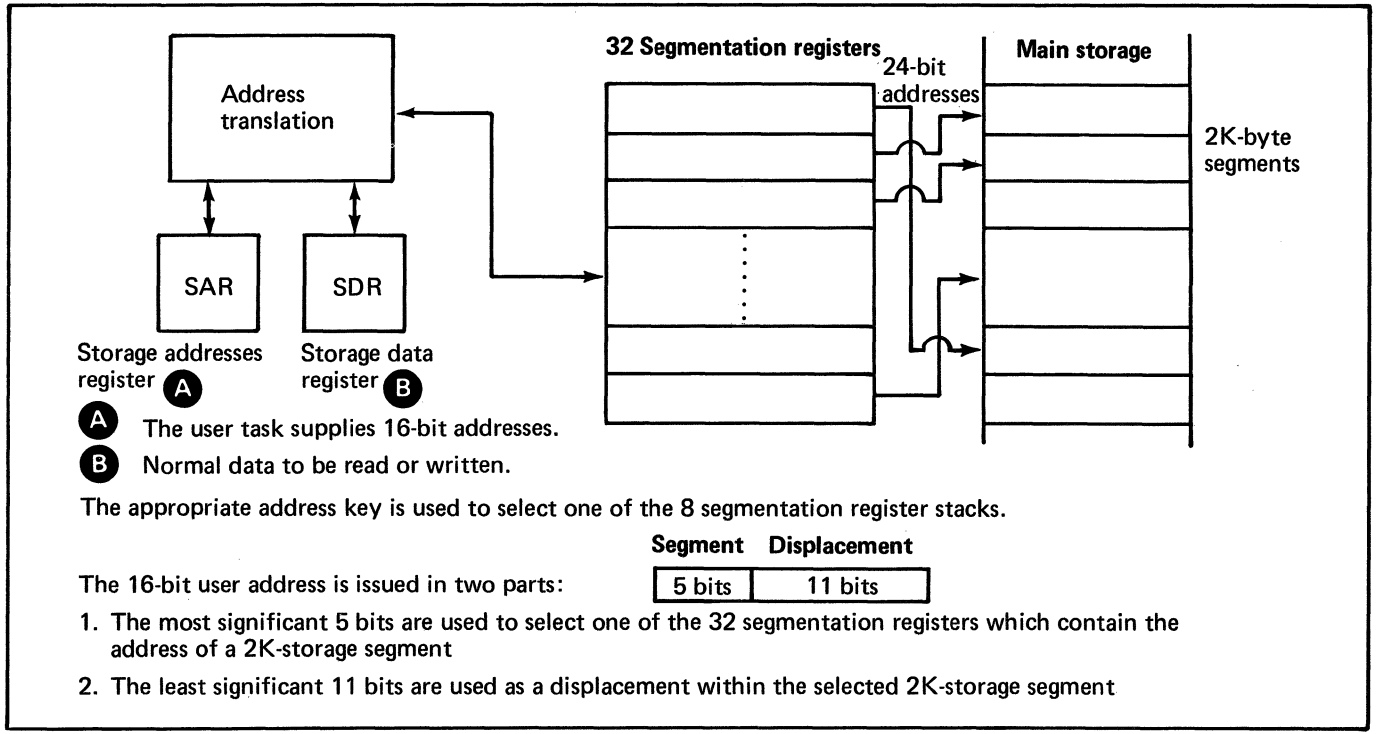
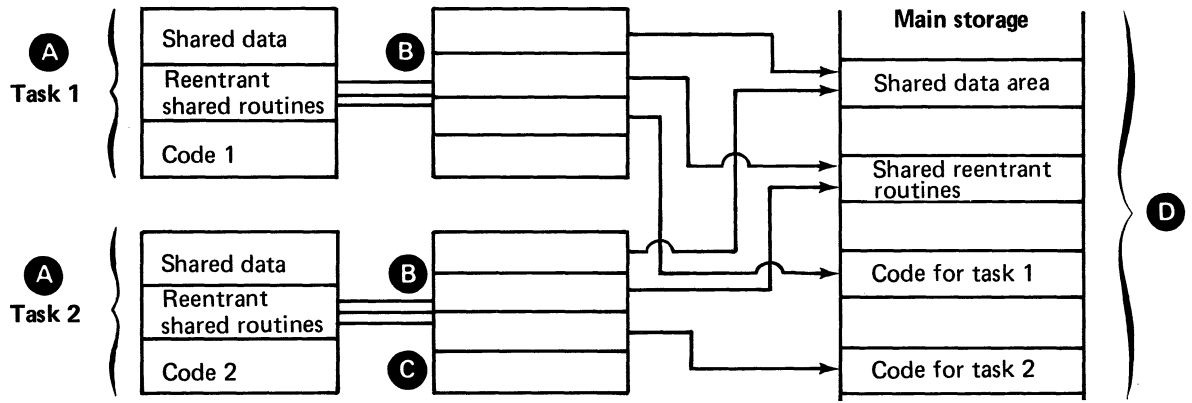


Figure 42. Conceptual basis for storage address translation



The actual Series/1 address mapping is both more complex and more general than shown in this illustration.

A Each task can have its own 64K-byte address space, addresses' code, data, and shared routines—just as if they were in a contiguous 64K-byte space.

B Segmentation registers relate physical storage locations to corresponding addresses in the user's 64K-byte address space.

C Setup and manipulation of the mapping addresses is a privileged operation usually performed by the operating system.

D The same physical areas of main storage may be mapped into each user's 64K-byte address space.

Figure 43. Conceptual mapping of main storage for two tasks sharing common data and subroutine areas

maximize hardware support of those small computer, software application requirements listed at the beginning of this chapter:

- Large, effective program size
- Fast task switching in response to events
- Extensive intertask communications
- Efficient implementation of intertask communications
- Thorough main storage protection

Mapped Storage Protection

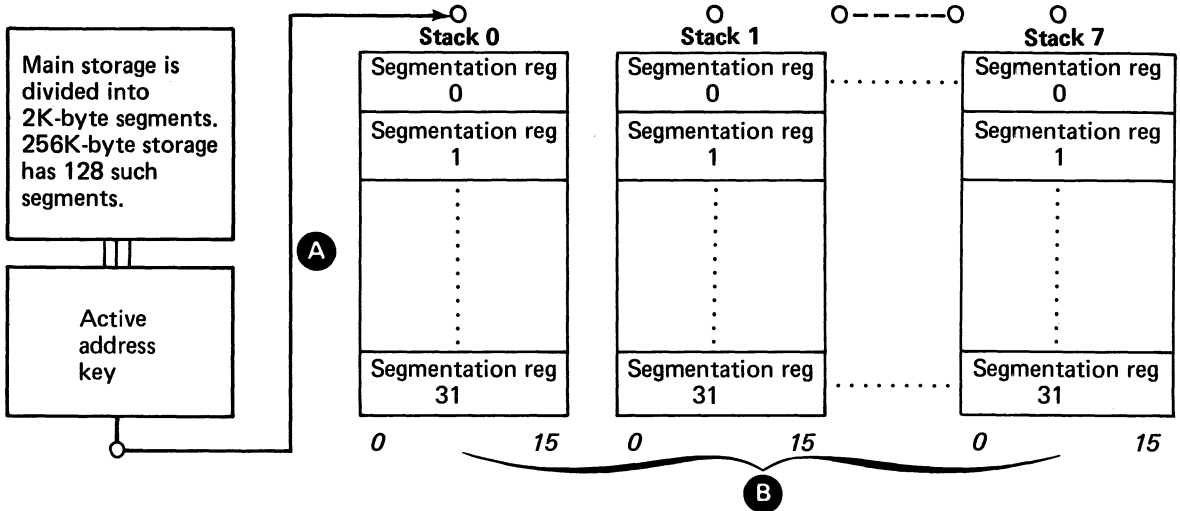
To enlarge effective program size and to enhance main storage protection capability, the concepts introduced as part of key-based, storage protection are extended to the translator. The Series/1 provides three address spaces to a task:

- One, for instruction fetching or program storage (controlled by the instruction space key)
- Two, for data storage (controlled by the operand 1 and operand 2 keys)

Intertask communications are clearly enhanced by the ability of multiple tasks to include common routines and data in their individual maps. The system increases task-switching speeds by using multiple sets of registers. This provision of the Series/1 helps prevent bottlenecks in the loading and saving of segmentation registers.

Segmentation Registers

The mapping process is shown in Figure 44. Main storage is segmented into 2K-byte segments. Eight sets of segmentation registers (32 registers per set for a total of 256 segmentation registers) are provided in the translator. Each segmentation register contains a physical, storage page address of 13-bit length to which is added the 11-bit displacement from the user-generated, 16-bit address illustrated in Figure 42. The translator then generates a 24-bit main storage address (maximum address of 16 megabytes). Notice that the 24-bit maximum, main storage address is a feature of the Series/1



A One of eight stacks of segmentation registers is selected by the value of the active address key and is used to map 64K-byte user address space into actual physical main storage. Selection is made via the address keys which are used for storage protection in those processors without address translation.

B Multiple tasks can be simultaneously mapped in the eight stacks of the segmentation registers. This mapping obviates saving and restoring the registers' contents when switching from one task to another.

Figure 44. Mapping task address spaces into physical storage using multiple sets of segmentation registers

architecture which provides for compatibility with potential future developments in storage technology. At present, the maximum main storage which may be attached is 256K bytes.

User Address Spaces

As indicated in Figure 42, each user has a 64K-byte address space which corresponds to a 16-bit address and, consequently, needs 32 segmentation registers to cover the 64K-byte space. The user's 16-bit address is used as follows: 5 bits for the page address (which selects one of the segmentation registers); and 11 bits for the displacement. The system assigns segmentation register sets to user tasks with the same mechanism it used for storage protection in the smaller processors without main storage mapping. Each of the 8 sets of segmentation registers is assigned a 3-bit address (0 through 7 in value), and the address keys are used to select the currently active segmentation register stack.

As shown in Figure 44, the currently active address key selects one of the eight stacks of segmentation registers which, in turn, are used to map main storage into the user address space. Since a user has three possible address keys for this task (Figure 45), three different 64K-byte address spaces can be mapped for one task.

The storage protection process is, of course, different when a Series/1 processor has a translator installed. Through a privileged instruction, the system may enable or disable the translator. When disabled, the system functions just as if the translator were not present. In that state, only 64K bytes of main storage may be addressed and storage protection is identical to that described earlier. When the translator is enabled, however, all protection proceeds by using the segmentation registers and the translator rather than by using the storage protection keys. In the enabled state, these keys are used to relate the particular mapping used by a task to a set of segmentation registers.

Segmentation registers are 16-bits long but need only 13 bits to contain the segment's physical address. Two of the remaining three bits are used for access control: one, to indicate access or no access to the storage block; one, to

indicate read-only access or read/write access. These are also shown in Figure 45. Since a given task can only access those portions of main storage mapped into its segmentation registers, storage protection—from an access point of view—is complete. When the Realtime Programming System is used, it prevents application tasks from accessing these registers and inadvertently changing their contents. In a custom-designed system, the user must carefully control access to privileged instructions and the segmentation registers. With the read/write control, storage protection in the mapped system is just as thorough as in unmapped systems, but it is actually more secure because it depends less on the address keys for complete protection.

Protection Violations

If a task attempts to access a block of storage, a class interrupt occurs and the system indicates the problem by setting the invalid storage address bit in the processor status word. Similarly, an attempt to change the contents of a block marked read-only causes a class interrupt, and the protect check bit sets in the processor. In this way the error detection and recovery process is similar whether or not the processor uses the translation capability.

Input/output devices access main storage through the I/O channel; they are also subject to the translator mechanism. This fact is exceedingly important to system performance because if the accesses did not operate in this manner, cycle steal operations would have to be done in those sections which correspond to contiguous main storage. This procedure is discussed further in Chapter 5.

Intertask Communications

Intertask communications of various sorts are a primary user concern because the structure of application needs in a small computer system consists of a set of cooperating, communicating tasks. As indicated earlier in this book, the organization and management of main storage is crucial to successful applications. These communications occur at

Each user task has three access keys:

1. ISK—instruction fetch space
2. OP1K—operand 1 key space
3. OP2K—operand 2 key space .

Each access key selects a stack of segmentation registers when it is active. Thus, each user task can actually utilize three 64K-byte address spaces. These spaces may be the same or different depending upon the choice of key values.

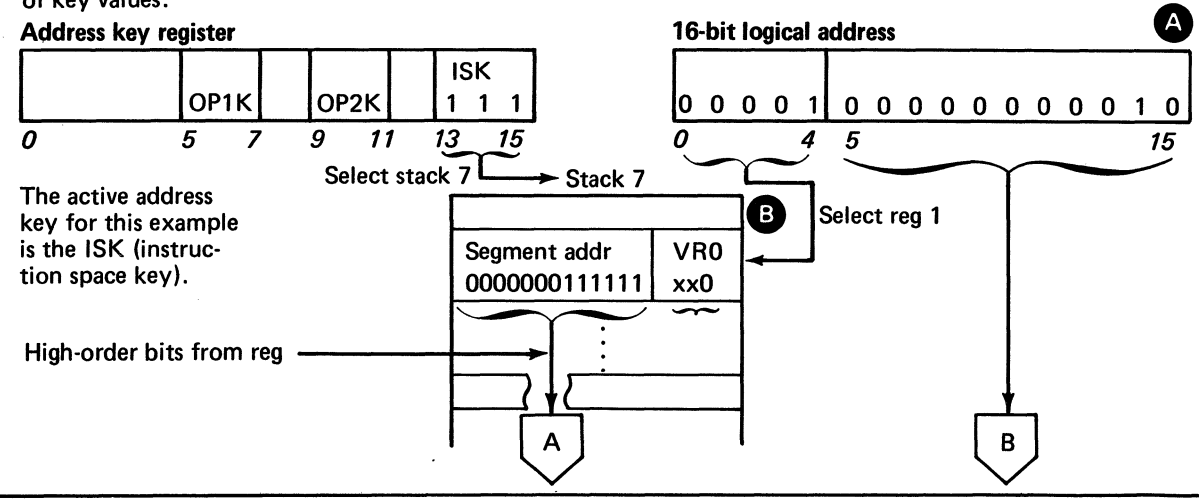


Figure 45. Multiple address keys for each task (1 of 3)

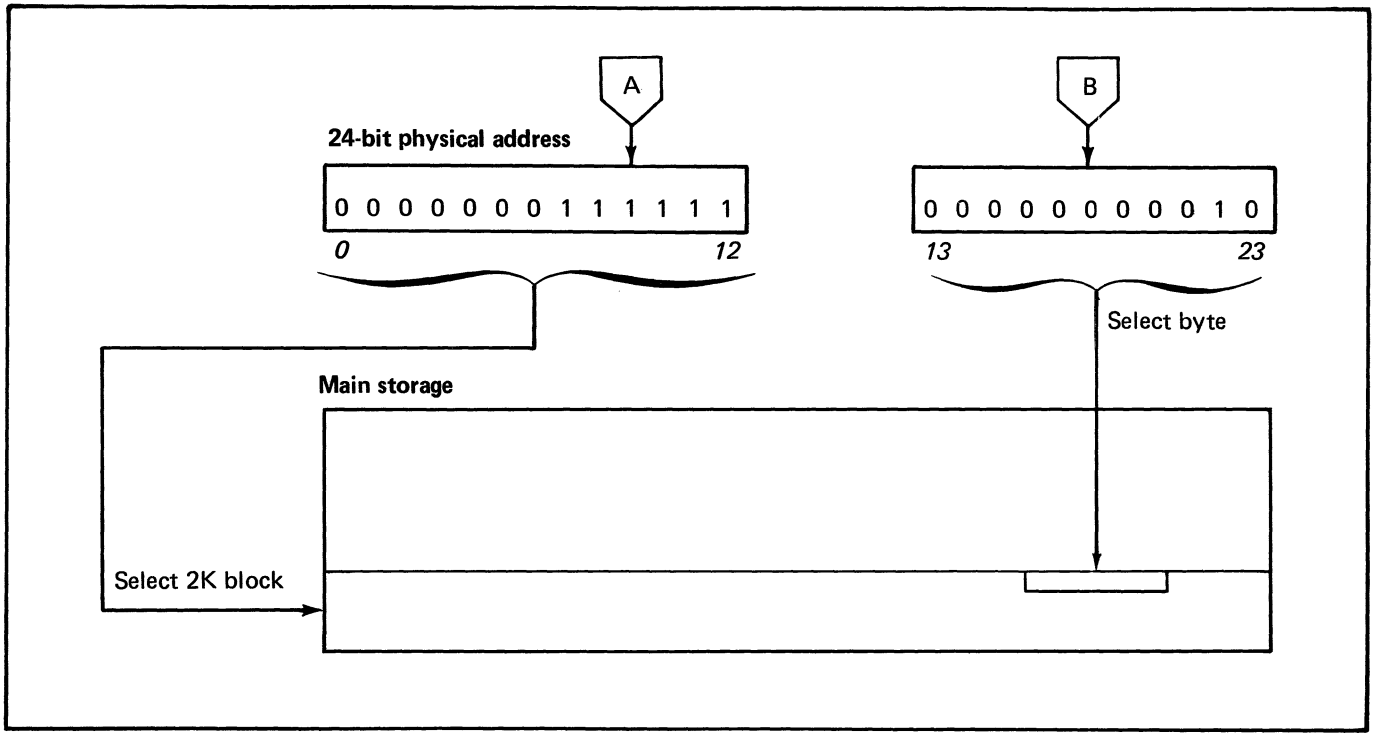


Figure 45. Multiple address keys for each task (2 of 3)

Ⓐ Address translation proceeds by:

1. Selecting a stack of segmentation registers using the value (one of eight) of the active address key
2. Selecting one of 32 segmentation registers using the five most significant bits of the user's 16-bit address; this address has been generated during an access to storage
3. Generating the physical storage address of the 2K-byte segment from the contents of the selected segmentation register
4. Using the least significant 11 bits of the user-generated address to select the particular byte within the physical storage segment

Ⓑ The 16-bit segmentation registers need only 13 bits for selecting physical storage segment addresses. This is so because the total physical address is 24-bits wide—with 11 bits being taken from the displacement within the 2K-byte segment.

The remaining bits are used for protection as follows:

V: segment is valid or invalid; if invalid no access is allowed at all which is useful when areas less than 64K-bytes in length are to be mapped

R: read-only flag

0: not used and must be zero

Figure 45. Multiple address keys for each task (3 of 3)

various levels, illustrating how the extensive storage hardware of the Series/1 can be integrated with the system software.

Tasks and the Operating System

In some applications, input/output operations on a specific device or set of devices can be dedicated to only one task. More generally, tasks share input/output devices; it is the responsibility of the operating system (standard or custom-designed) to prioritize contention for the devices, and to perform most of the detailed aspects of the input/output operation. This might be done by enabling those subroutines, which a task calls, to:

- Request an input/output operation
- Check the status of a device
- Perform some code conversion
- Carry out additional operations

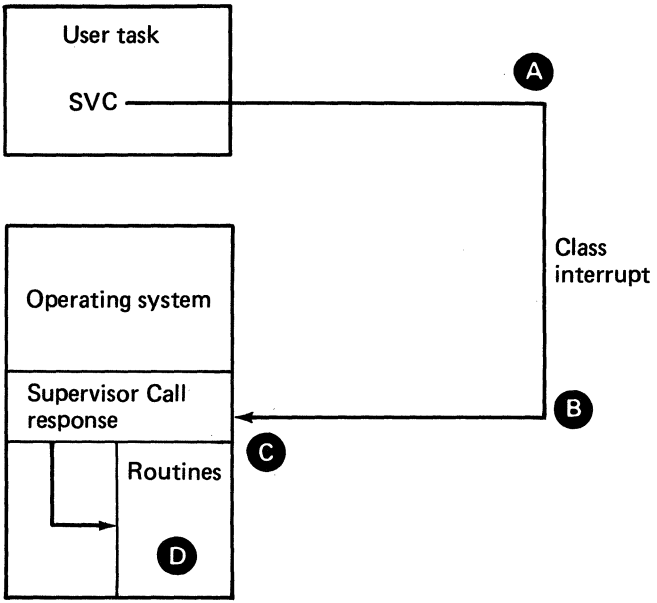
This approach to task communications can be disadvantageous because the subroutines involved are shared and—unless the main-storage management is exceptionally well designed—they may consume part of the user's address space. As an alternative method, all input/output operations and other executive facilities may reside with the operating system; communications between the user task and the operating system are performed via an internal interrupt designated as a supervisor call.

As shown in Figure 46, the user executes a Supervisor Call instruction which transfers control to the operating system. The operating system first analyzes the parameter supplied in this instruction to determine the type of request the user is making, and then carries it out.

Because there are so many individual services associated with input/output data management and scheduling, the user should consult the Realtime Programming System documentation for specific details.

Tasks and Separate Data

It is a characteristic of realtime, small computer applications that tasks access data areas separate from the task itself for



A To invoke any operating system service (including input/output), or any operating system routine, a Supervisor Call is executed.

B A Supervisor Call generates a class interrupt which is responded to on the same, rather than a higher, hardware priority level. The Supervisor Call mechanism provides complete system integrity as well as the maximum usable address space for the application task.

C The interrupt response routine saves the level status block of the application task and then carries out the routine requested within the address space of the operating system. Upon completion, the user's registers are restored and control returns to the user's application task.

D Routines within the operating system do not have to be part of the user's address spaces; this fact allows the user to prepare larger application programs.

Figure 46. Communications between an application task and the operating system via supervisor calls which generate a class interrupt

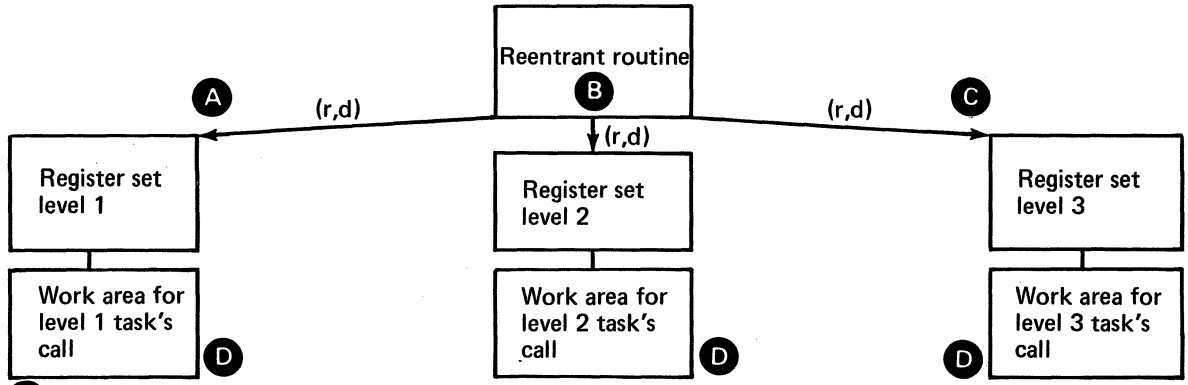
one of two reasons: 1) the task needs to be reentrant or; 2) the task needs to share routines or data bases. This access is not difficult if addresses are known at program preparation time, but it is complex if addresses are not known until execution time. Using addressing modes for main storage simplifies the access operation.

Figure 47 shows a routine accessing several different areas using register displacement addressing mode. In the figure, the calls to that reentrant routine are assumed to be on different hardware priority levels so that each call has its own copy of register 3 in hardware (user registers are duplicated on each priority level). Consequently, the code can refer to any item by its displacement, relative to the address in register 3; depending upon which level is active, the address will refer the code to the correct work area. If the calls are from the same level, it is necessary to save and restore only the contents of register 3 (or whatever register is used). The important point in this procedure is that the reentrant code itself does not have to be concerned with the location of the particular work area. No interrupt disabling or similar functions need occur during the execution of reentrant tasks.

Task Switching

Frequently, the most economic design of small computer applications is achieved by dividing the application into many concurrent tasks, each of which is relatively small and well defined. Similar economic advantages have been demonstrated by structuring individual programs. If, however, the management of these concurrent tasks is cumbersome, the advantage of achieving fast and reliable response to events is lost; in these circumstances, users might have to combine the tasks in one program so they can perform scheduling and task switching at program preparation time.

The Series/1 architecture is designed to expedite task switching with minimum overhead. As explained earlier, each hardware priority level of the system has its own set of user registers whose contents are duplicated in hardware. This means that when the system switches from one interrupt



A shared routine is called by the tasks on different priority levels. The shared routine accesses data within the work area using base relative addressing. On each priority level, duplication of registers in hardware insures that there is no conflict between multiple calls.

Figure 47. Addressing modes facilitate reentrant routines' use of multiple work areas

level to another, it need not either save or restore the register contents. In a small system, the designer can insure fast response to events by carefully allocating tasks to different priority levels. The system responds to internal or class events on the same priority level, but, in order to make the response more rapid, the hardware itself also assumes the responsibility of saving the level status block.

Generally, context switching becomes more difficult in large systems. However, the Series/1, by providing eight sets of mapping registers for large storage systems, allows a set of tasks—which must respond quickly—to remain mapped as shown in Figure 48. When an event occurs (priority interrupt, for example) and the interrupt response tasks have been previously mapped, the system need not change either the level status block registers or the segmentation registers. This resource gives the Series/1 users an important level of control over their systems. By allocating tasks to storage and premapping each task's segmentation registers, users can control the response time to events occurring in their application.

This control is most valuable to an OEM user customizing a software system. To insure that, at execution time, execution can begin with a minimum of overhead, the user must set up in advance the following relationships:

- Tasks and groups of tasks (task sets)
- Shared data areas
- Shared routines
- Storage addresses

The Series/1 Realtime Programming System fully supports this level of control over tasks, thereby permitting a more effective structuring and implementation of small computer applications.

Auxiliary Storage Management

If all user-application tasks are permanently resident in main storage, the system can respond rapidly as outlined in the previous section. However, many applications require

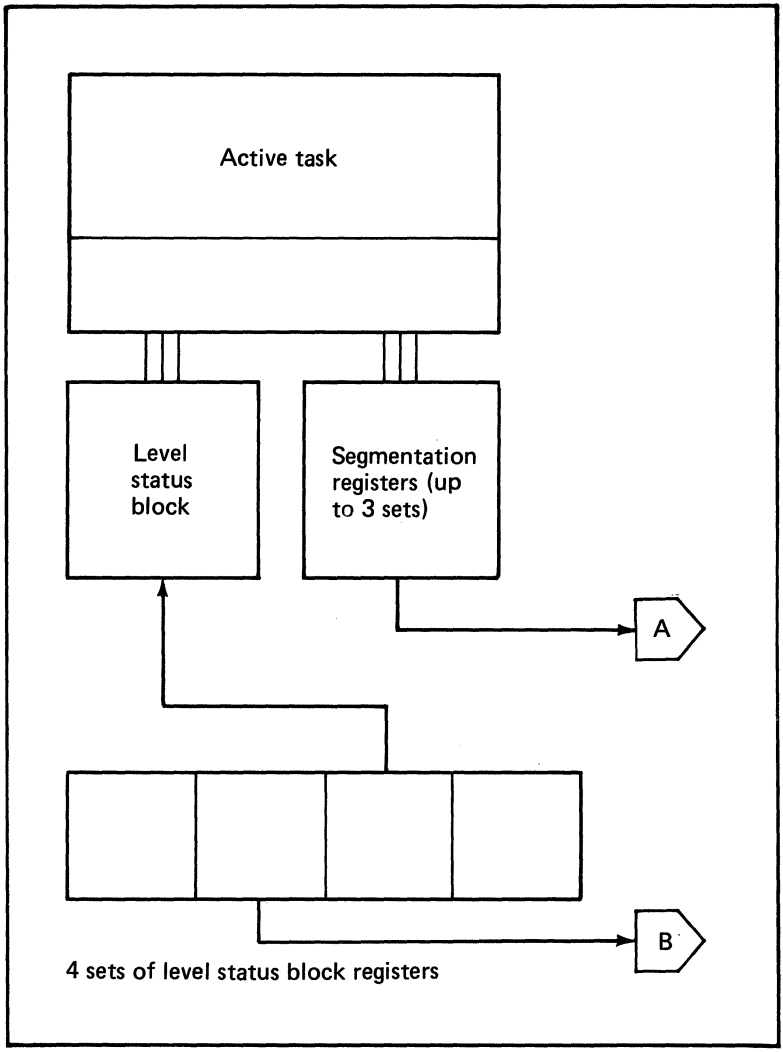


Figure 48. Context switching (1 of 3)

secondary or disk storage because the application consists of a large number of tasks. Response time of disk-resident tasks cannot be as fast as that of tasks resident in main storage; however, the response time must be relatively rapid and be carried out under control of the application designer.

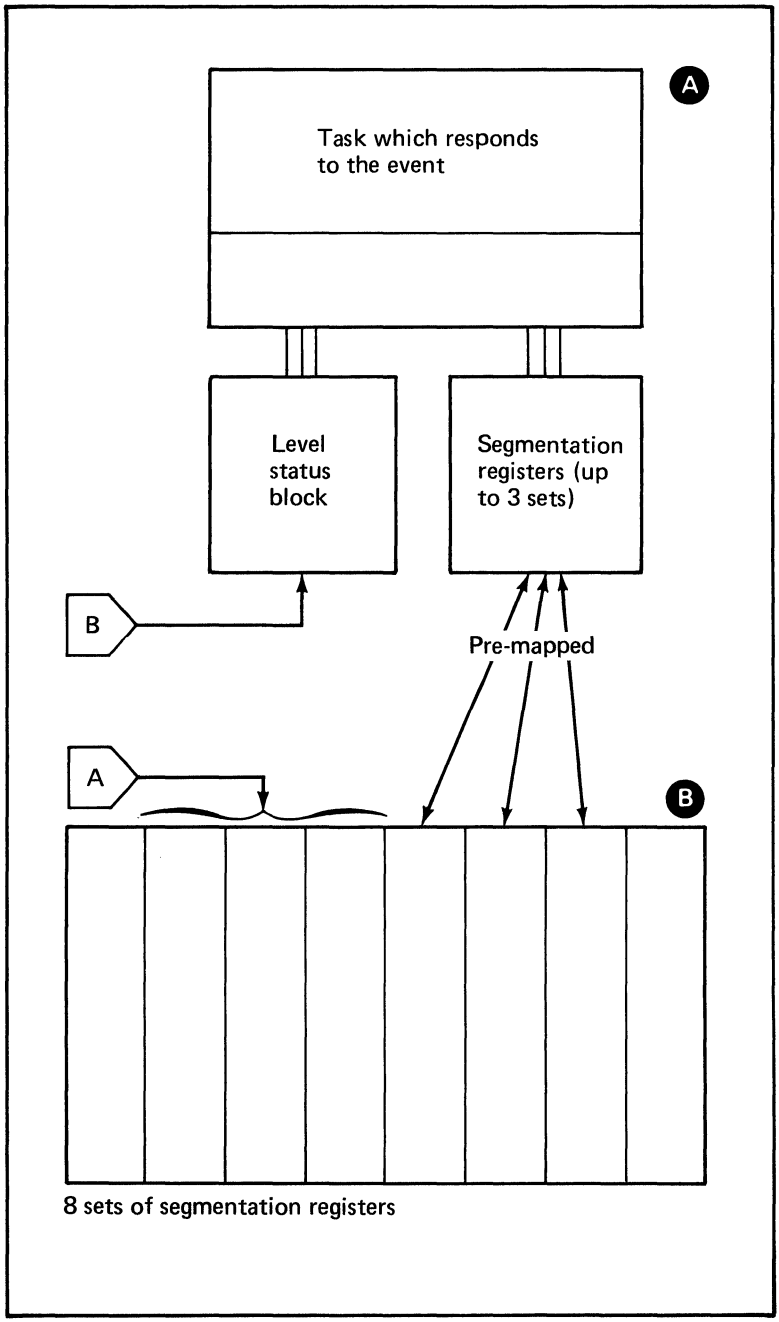


Figure 48. Context switching (2 of 3)

(A) To respond to the event, it is necessary that the contents of the active task's level status block registers and segmentation registers not be destroyed; this will insure that the active task can resume executing at a later time.

(B) If the event response task were to use the same address keys and need a different mapping from that of the active task, it would be necessary to save the segmentation register's contents and load them again with the mapping addresses for the event response task. This is a time-consuming operation (almost 200 microseconds).

With eight sets of segmentation registers, however, fast response tasks can be permanently mapped into the segmentation registers. Consequently, there is no mapping overhead when responding to the event.

If tasks respond on different levels, the level status block need not be saved or restored because these registers are duplicated on each level.

The net result: *very fast context switching to respond to events.*

One of the design problems in small computer applications is the choice of which tasks are to be resident, which are to be mapped, and on which levels they are to execute. When these design problems are carefully resolved and implemented, the Series/1 hardware provides fast response times.

To start or resume execution, a task must have its registers (level status block) initialized or restored to their previous status; the registers must also be mapped.

Segmentation registers associated with each key used by the task must be loaded with proper physical storage addresses and read/write control information.

Context switching can be defined as: the change from one active task to another in response to some event; the rapid response assumes that the overhead involved in the switch is not excessive.

Figure 48. Context switching (3 of 3)

Storage management in the Series/1 is a user option with the Realtime Programming System.[1] The system can take advantage of hardware address translation to load tasks into 2K-byte segments—wherever they are available—instead of loading them into contiguous locations. Such storage management is termed "dynamic" because the system discovers space for data or tasks whenever the task is to be loaded and wherever the space is available. The Series/1 hardware permits this type of management. In fact, once the system loads the segmentation registers with the storage segment addresses, the cycle stealing, input/output system—using the translator—can load the program from the disk into the non-contiguous storage blocks.

The Series/1 also offers a partitioned storage management because:

1. Rapid response to inquiry requires residency in main storage
2. The real limitation on response to disk-resident tasks usually involves the amount of program code and data that the system can transmit to and from disk

This partition system is essentially fixed in the sense that partitions are set up at system generation time. Dynamic partitions are useful for those tasks which are not time-critical and which can remain resident until they are completed. To prevent interference with the response of the tasks in the fixed partitions, the dynamic partitions are kept separate from the fixed partitions.

Figure 49 shows this storage management system. Individual partitions contain one task set at a time. A task set is a group of tasks which:

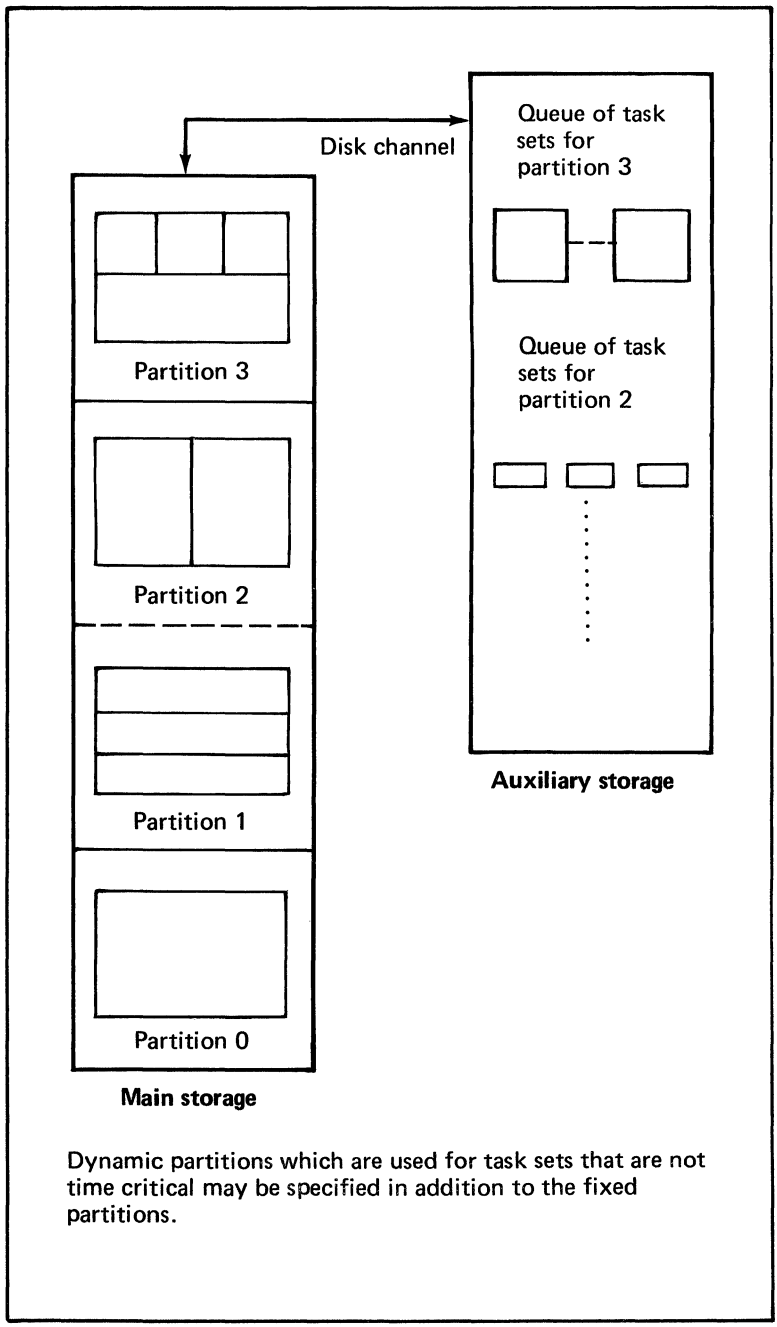
- Execute concurrently
- Communicate extensively among themselves
- Are prepared as a group for loading into a partition

[1] Storage management is often termed "memory management" in small computer literature. The term "storage management" is used here to differentiate between main and auxiliary storage.

- The Realtime Programming System uses a fixed partition organization for main storage in order to expedite: 1) the operation of those applications involving a set of cooperating tasks and; 2) user control over event responses.
- A task set is a group of tasks loaded as a unit from auxiliary storage. Only one task set at a time resides in a partition. Once resident in a partition, task sets remain there until they complete execution. Optionally, a partition may roll-out one task set in order to bring in a higher priority task set; this is usually done only for a background task.
- There may be up to 16 fixed partitions, one of which is occupied by the operating system. Partitions may be of any size—in 2K-byte increments.
- Fixed partition main storage management was selected for the Realtime Programming System because:
  - There is less swapping of tasks; once in main storage, most task sets stay there until they complete execution. This principle reduces traffic on the disk channel which helps prevent a common bottleneck in small computer systems.
  - There is less overhead involved in saving and restoring values in segmentation registers
  - There is much more user control over the system and its response-time delays; this control permits the user to assure adequate response in critical applications through proper layout of partitions, tasks, priority levels, and access keys
  - It is easier for the user to set up and control access to shared routines, shared task sets, shared data areas, and system-wide data bases

Figure 49. The Realtime Programming System storage management (1 of 2)

As indicated earlier, task sets can communicate with the operating system through supervisor calls, and can communicate with other tasks in other partitions if the user indicates this communication at program preparation time. Clearly, the ability of the storage management system to map non-contiguous areas provides the mechanism for hardware support of the very difficult, intertask communications' procedure.



Dynamic partitions which are used for task sets that are not time critical may be specified in addition to the fixed partitions.

Figure 49. The Realtime Programming System storage management (2 of 2)

Task sets in a partition usually remain there until they complete their execution. If they did not do so, the system would have to transmit them through the cycle steal channel multiple times and, consequently, overload the channel. Optionally, however, a partition may permit the roll-out of a task set to accommodate a higher priority task set waiting to execute. Even in this case, the roll-out is limited to one task set. Experience with fast-response, large systems indicates that application designers can assign task sets to partitions in such a way that response time is adequate as long as the disk channel does not saturate the system with a long list of requested transfers. An important additional consideration is that the application designer can control the response time to external events through assignments of task set priorities and partitions.

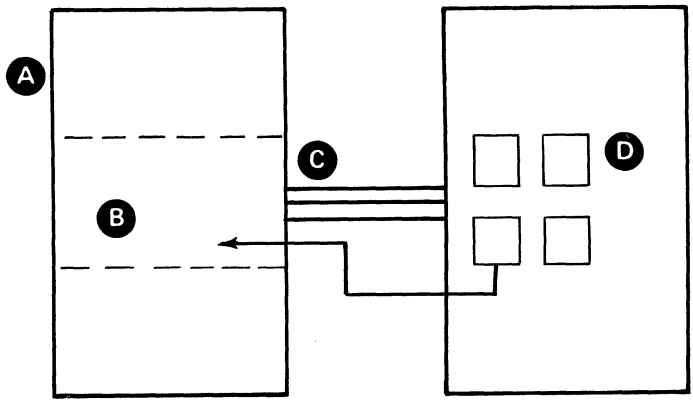
As needed, dynamic partitions are set up for task sets which are not assigned to execute in a given fixed partition. This procedure is essentially equivalent to allowing one of the partitions in Figure 49 to contain several task sets simultaneously. The Realtime Programming System allocates space in the partition on a first-come, first-served basis. In this way, multiple task sets may occupy the dynamic partition area. Again, note that this allocation feature increases the amount of work required by the operating system to initiate a task and, consequently, slows the response time. Nonetheless, the combination of fixed partitions, with specific task sets assigned to them, and dynamic partitions, with their inherent flexibility, provides system implementors with the tools needed to create the small computer software organizations that best fit their applications.

Storage Overlay Management

Often, a dedicated application requires access to a relatively large data area which does not fit well into the address space. This can occur even though Series/1 systems with hardware address translation have three address spaces available. In this case, it is possible to use storage overlay to solve this problem. Figure 50 shows a task within which is allocated an area for the data base, but the area is not

The standard overlay scheme permits programs to be larger than the allowable address space—provided they fit in physical main storage. A program too large for storage or address space reserves one area within itself which is used to contain routines or data areas (one at a time, as needed). When needed or called, the routine or data area is transferred from auxiliary storage to the area within the task, and "overlays" the routine or data area previously there.

Standard overlay scheme



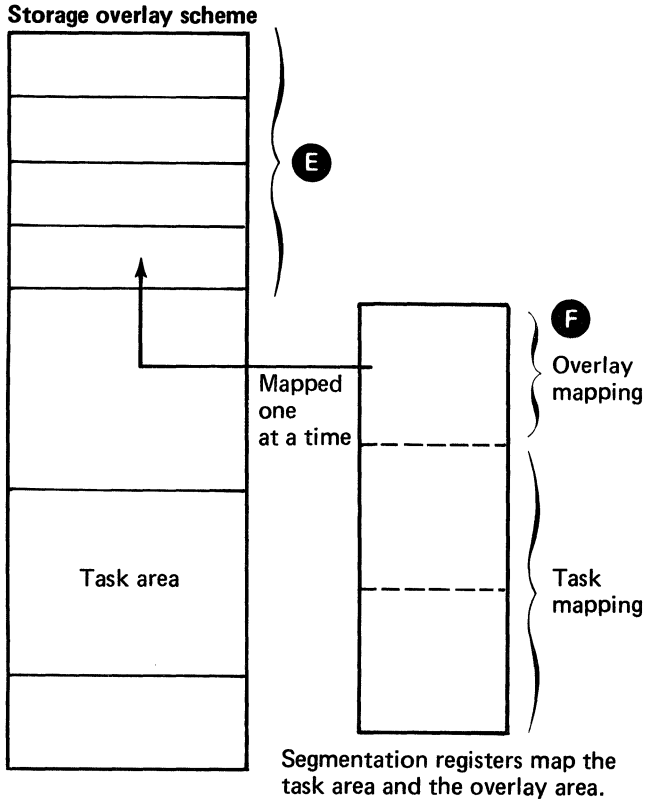
A Task area

B Overlay area within a task

C Disk channel

D Overlay segments (routines or data sets) loaded on call

Figure 50. Overlay methods of storage management (1 of 2)



E The overlay units reside in physical main storage rather than on the disk.

F The overlay area within the task's address space is *remapped* by a call to a special privileged routine whenever another overlay segment is desired. The time required is the time needed to load several segmentation registers (about 5 microseconds each), rather than disk transfer time.

Storage overlay is effective when the task is too large for the address space (even when using three address spaces with three different keys); the task will reside in physical main storage.

Figure 50. Overlay methods of storage management (2 of 2)

large enough to take the entire data base at one time. The entire data base is loaded into main storage with the task, rather than being transmitted from the disk each time a segment is needed. To overlay the program area with a segment of the data base involves changes only to the mapping registers. No movement of the data base occurs. As indicated in Figure 50, a privileged routine is called by the task which then loads the appropriate addresses into the segmentation registers. Custom-designed systems can use this technique effectively to control and facilitate larger, data management-oriented applications. Both the Program Preparation System and the Realtime Programming System also support storage overlay use of the hardware translation feature.

The overlay concept can be used for shared routines in the same way. Even if fast-response systems do not fit the 64K-byte multiple address space provided in a mapped machine, they can be effectively accommodated by the Series/1. The combination of main storage hardware and software designed into the IBM Series/1 supports both large and small applications in a controlled, responsive manner.

5 Organization and Management of the Input/Output System

Important Factors in Computer Input/Output

The organization of input/output is critical in any small computer application—especially when throughput, reliability, error detection, and compatibility with OEM devices and peripherals are considered. Because applications vary so much in size and complexity, it is necessary to integrate the input/output hardware into both the processor and the software. Otherwise, the system cannot, simultaneously, meet all of the requirements listed in Chapter 1. Figure 51 shows the various levels from which a user can view the input/output system. At the lowest level in the figure—the device level—the system concerns itself with:

- The self-diagnosing capability of the device and its interface *and* their consequent easy maintenance
- The ability to perform extensive error detection within the device and its interface to the processor
- The capability of operating under complete processor control when input/output volume is low
- The capability of off-loading the processor when input/output volume is high

In other words, the system must be so designed that the volume of local intelligence placed in the device and its

interface depends upon the device and the amount of input/output it performs. As indicated earlier, the Series/1 interfaces incorporate microprocessors which provide this flexibility of data flow together with the self-diagnosing feature. Since the functions performed by the interface are so device-dependent, they are discussed later in this chapter when the devices themselves are considered.

Processor Level

The processor level of input/output control varies considerably from computer to computer. It is important because it affects interfacing, software at the lowest level, and the overall organization of an application. Both the direct program control and cycle steal data transfer capabilities are necessary: the former is appropriate for slower devices where the low volume of data does not impact system throughput; the latter is necessary for high-volume and high-speed data transfers. Cycle steal data transfer also facilitates the offloading of the processor. As the system places more and more input/output control functions in interfaces, it usually transfers more and more data before involving the processor. The cycle steal mechanism reads and writes this data directly into main storage where it is ready for processing when the transfer is complete.

The Basic Software Level

The first two levels shown in Figure 51 provide the basic capability for input and output in the system hardware. The utilization of that capability through software processing is vital to the system's operation. The basic software level is the interface between application tasks and the input/output system. Applications running under control of the Realtime Programming System need the same control over devices that they would have if they were operating at the processor level with completely customized software. Otherwise, critical applications could not realize the full machine potential. Similarly, applications which do not use the Realtime Programming System still need control over tasking and input/output.

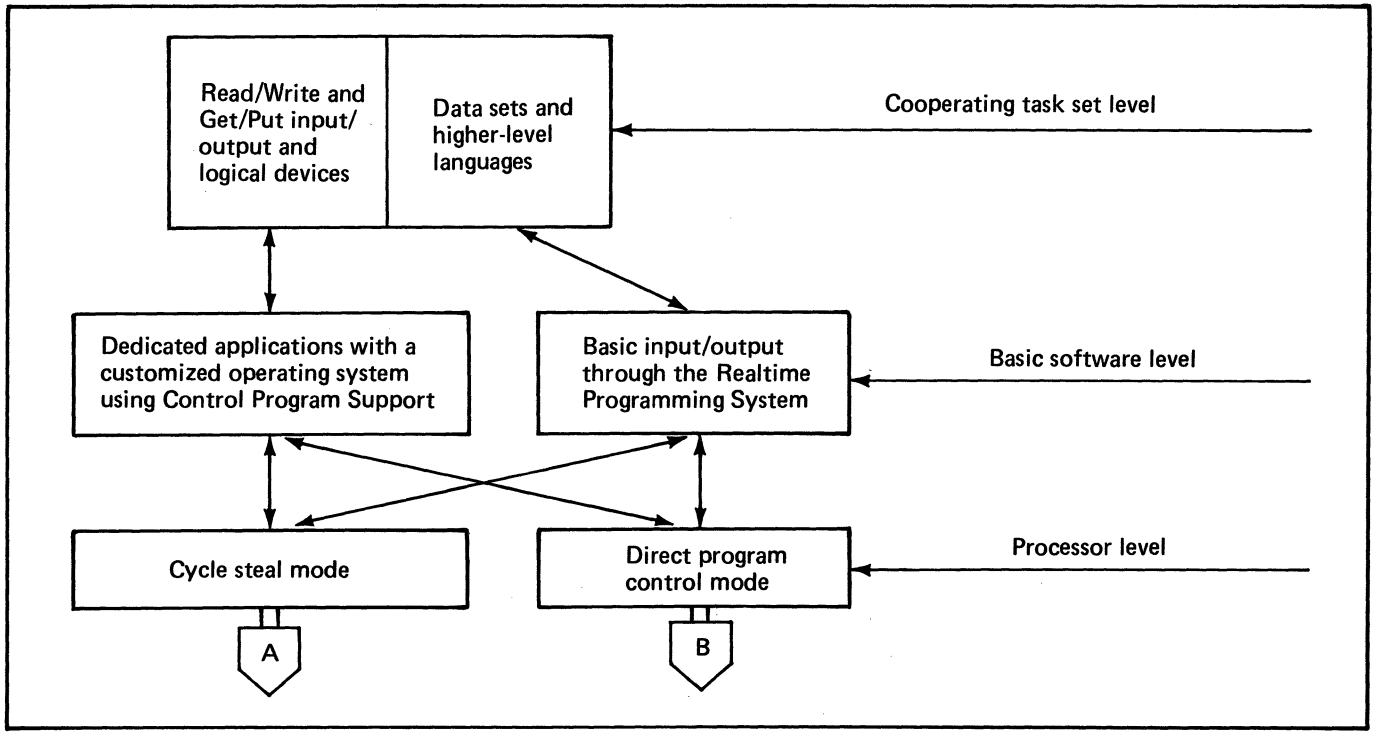


Figure 51. The levels from which input/output must be considered (1 of 2)

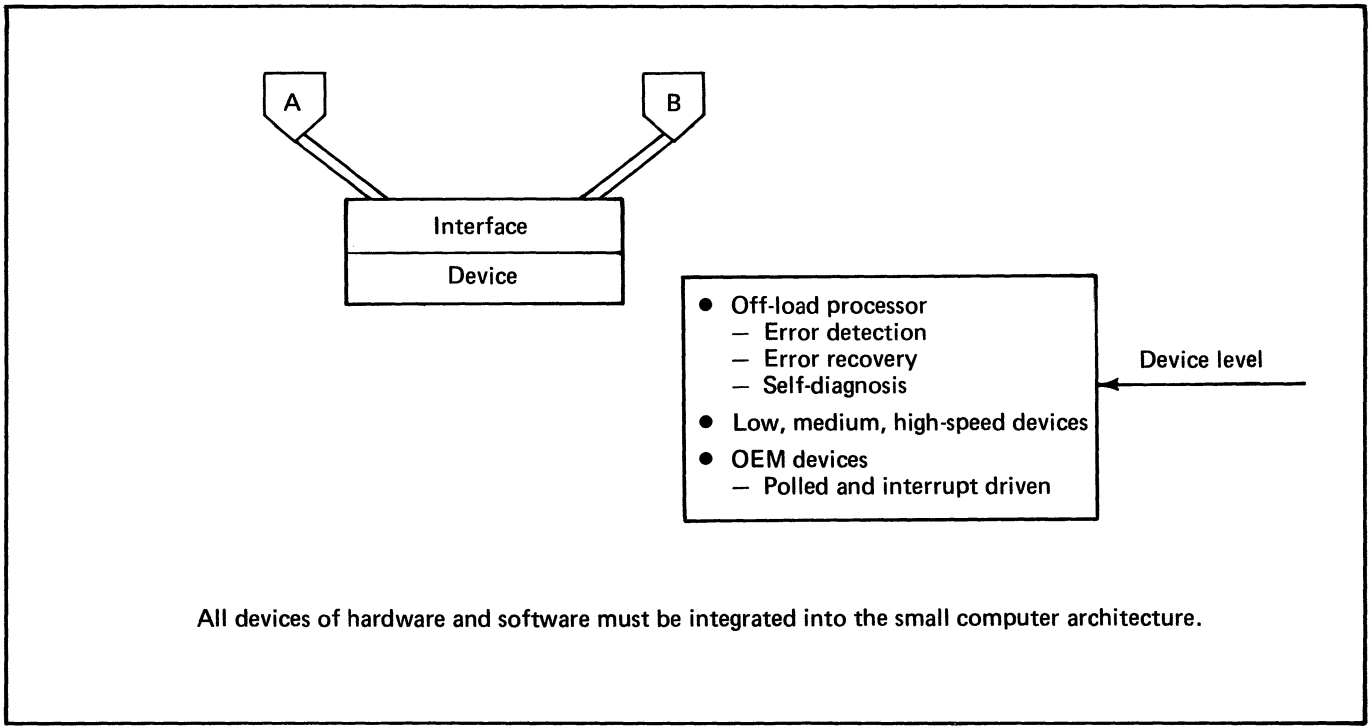


Figure 51. The levels from which input/output must be considered (2 of 2)

Users must take into account the limitations in device control imposed by software systems which were not designed to be integrated with hardware. When hardware/software integration is not accomplished, it is conceptually but not practically feasible to write custom software because the careful, manufacturer-designed interactions between modules of the software are typically unavailable to the user. For example, error detection and recovery is just as important in a small computer application as is the input/output itself. Error detection and recovery is a cooperative effort among the hardware and software modules; deleting part of this integrated software—and replacing it by special purpose software—may solve a particular application problem while, simultaneously, losing many of the overall advantages the system provides. Both the Control Program Support package and the Realtime Programming System provide a complete control over input/output that is equivalent to the control obtainable at the processor level. As a result—without sacrificing self-diagnostics or extensive error checking—users can devote their time and efforts to the application itself rather than to the system software design.

The Cooperating Task Set Level

Many small computer applications are themselves small, with dedicated, critical input/output requirements which are completely satisfied at the basic software level discussed above. Many other applications, however, need a more general level of support provided at the cooperating task set level shown in Figure 51. When a number of tasks execute concurrently, it is often desirable to provide a less complicated interface to the input/output system from the user tasks. The Realtime Programming System provides two control levels in addition to the basic software level; they are the Read/Write and the Get/Put levels of control. The Read/Write level presents a single physical block of data for transfer between the application tasks and the named device. All details of this transfer are handled by the system input/output software within the Realtime Programming System.

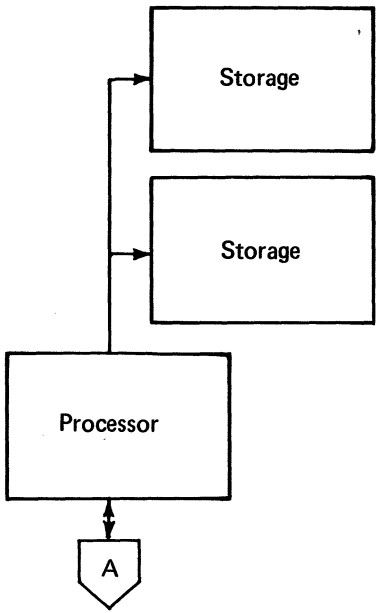
The Get/Put level is higher than the Read/Write level because it provides logical rather than physical block transfers. The system handles all transferrals between logical and physical blocks and device names. The highest level of input/output control is convenient for application tasks written in higher-level languages like FORTRAN, COBOL, or PL/I because it minimizes programming and debugging efforts.

It is not necessarily true that the higher the level of interaction with the input/output system, the less efficient is the use of the processor. In fact, the typical Series/1 application is realized as a set of cooperating tasks using application devices and sensors that require detailed control over buffer sizes, number of buffers, level of priority, and similar areas. The user may program these aspects of an application using assembly language macros provided with the system software. Once the control over the input/output system has been established, the user can access it efficiently through commands in the higher-level languages. The net result is a structured control over the input/output system.

This chapter discusses the input/output system and illustrates the structured control integrated into the hardware and software design.

Overview of the Series/1 Input/Output Channel

The Series/1 offers a single channel to which all input/output devices are interfaced including special processors like the floating-point feature. Details of the Series/1 input/output system may vary from processor to processor; the user should consult the appropriate processor reference manuals for exact details. Most of the specific examples used in this chapter apply to the larger processors like the 4955. Although different applications require different groups of devices, Figure 52 illustrates some possible combinations. The figure notes that the user may install more than one of each input/output device, if needed. Physically, each device is interfaced to the input/output bus via a printed circuit card which plugs into either the processor card file or an input/output expansion unit (Figure 53). The expansion units may require a repower card to provide adequate power and isolation.



- The Series/1 input/output channel is a bus to which all devices connect, including:
  - Those under direct program control
  - Those operating under cycle steal mode
  - Those functions implemented as parallel processors (like the floating-point feature)
- Up to 256 devices may attach
- Multiples of devices may attach
- The user sets the device address by using jumper selections on the device interface rather than by positioning the address within the enclosures

Figure 52. Input/output device combinations (1 of 4)

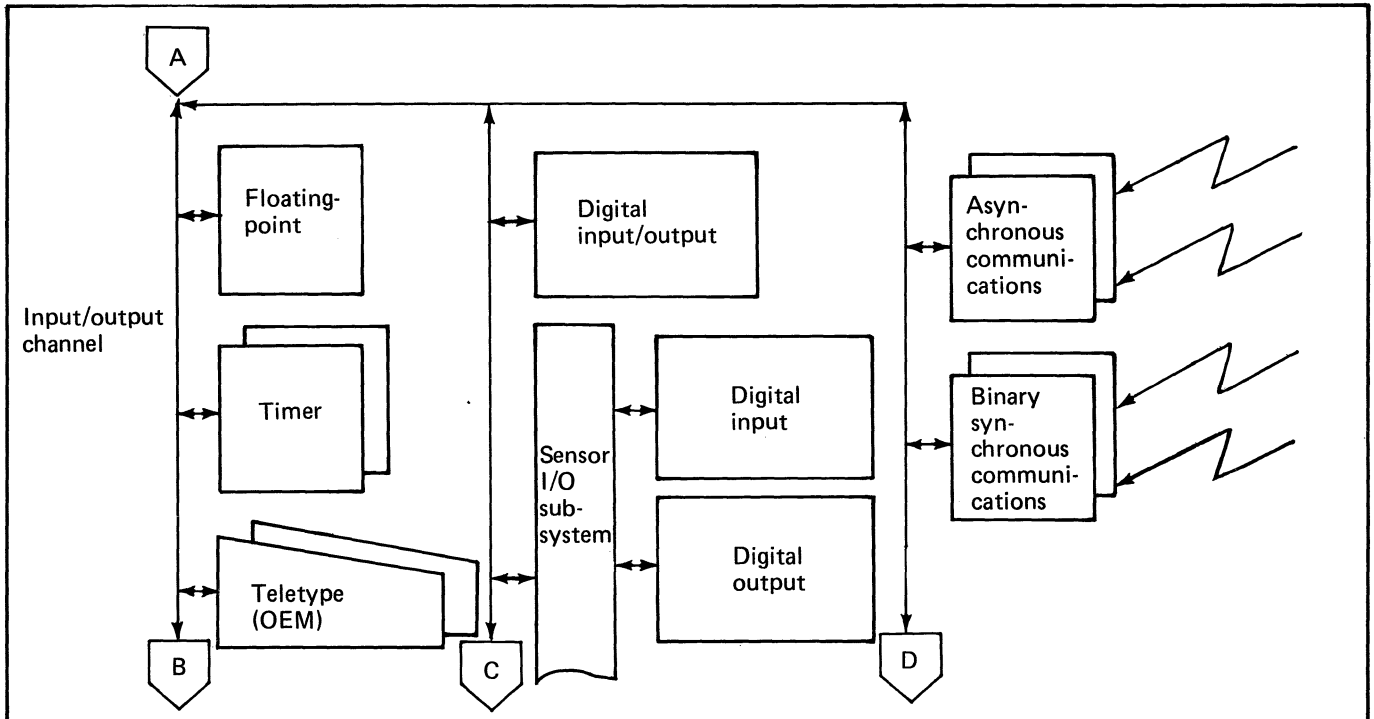


Figure 52. Input/output device combinations (2 of 4)

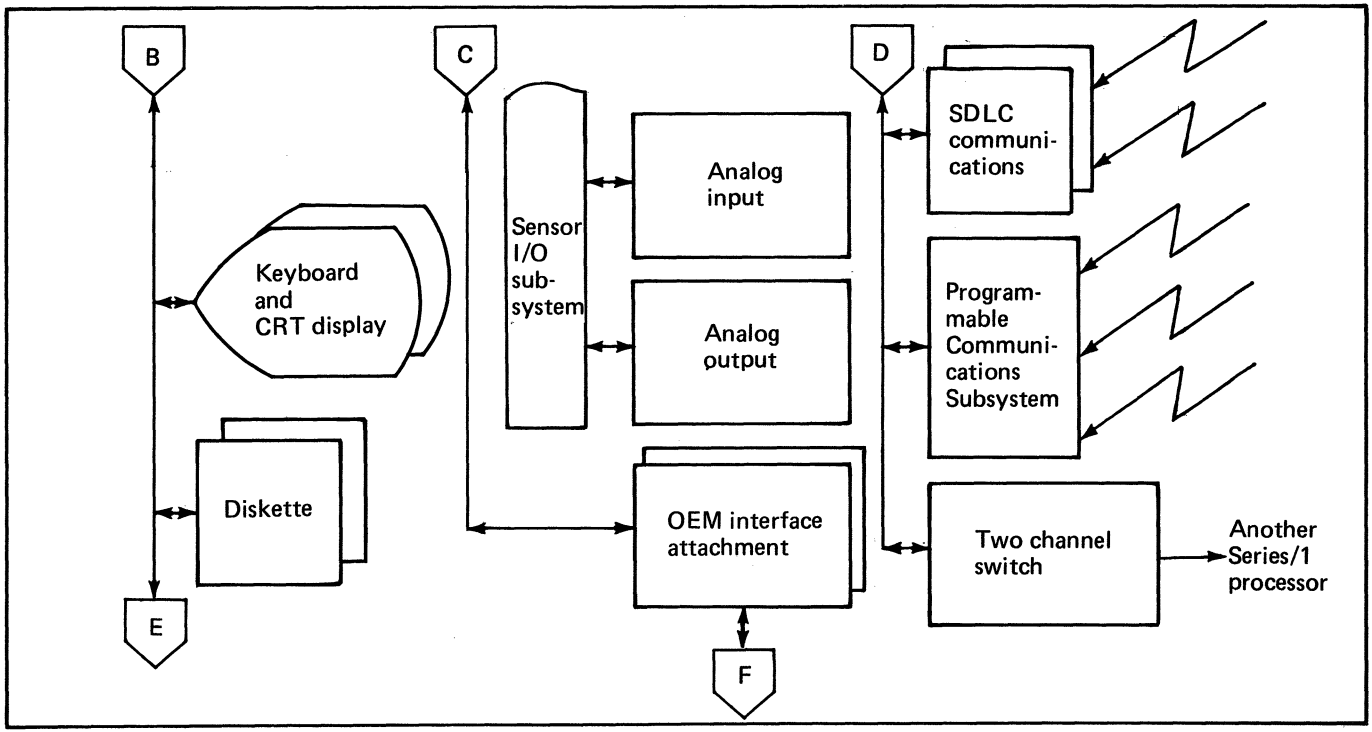


Figure 52. Input/output device combinations (3 of 4)

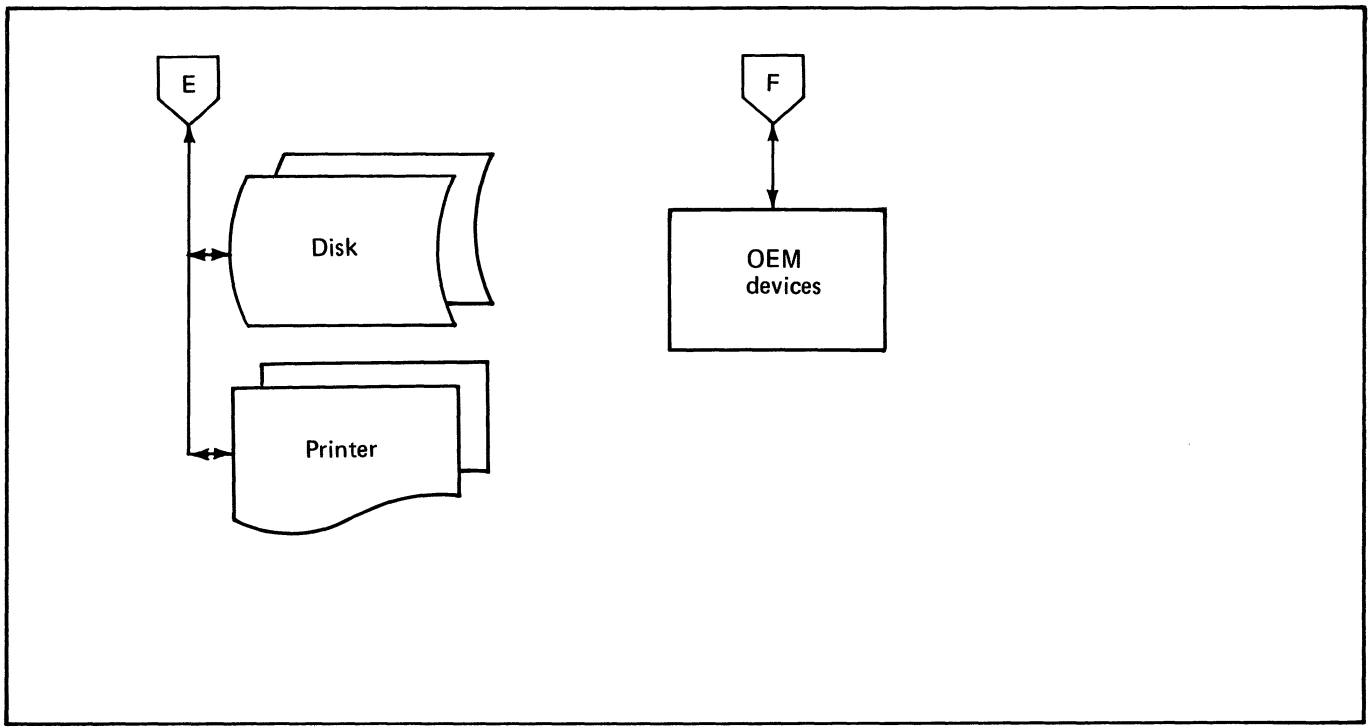


Figure 52. Input/output device combinations (4 of 4)

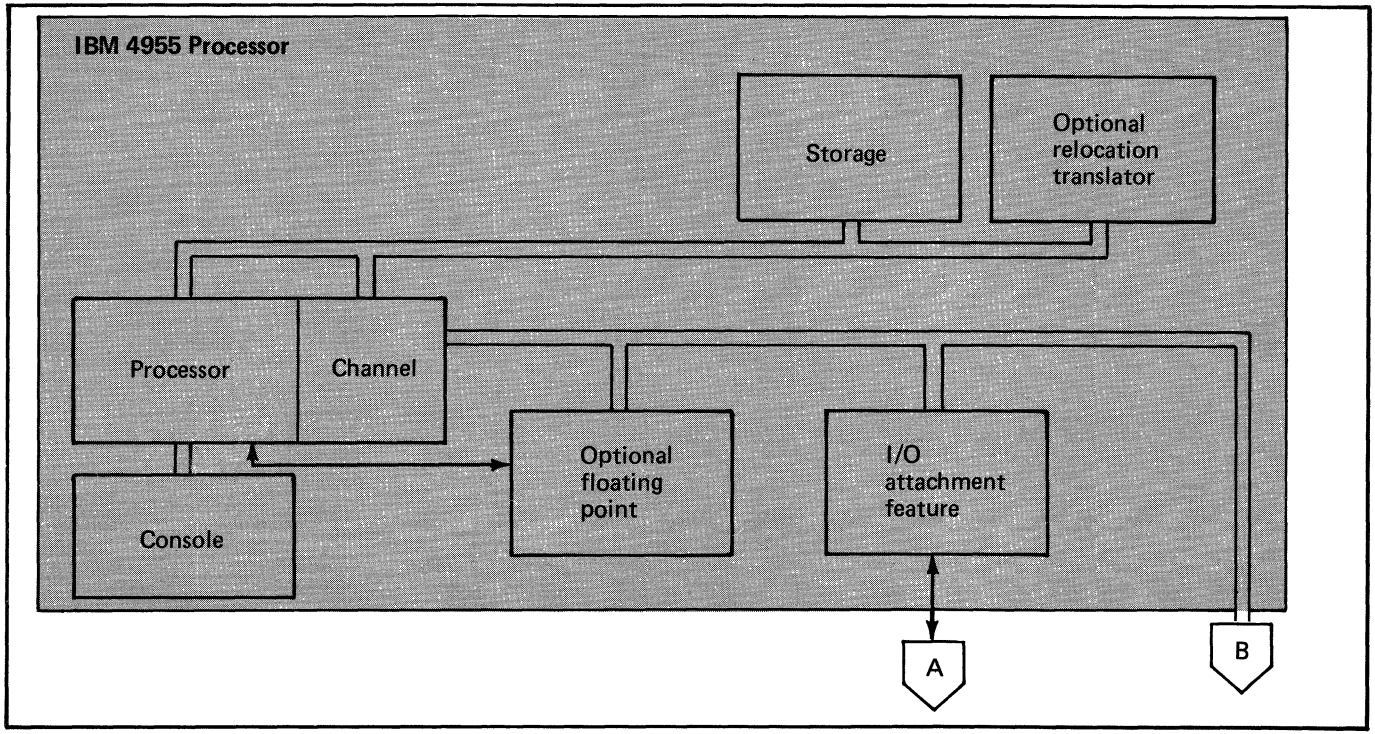


Figure 53. The Series/1 4955 Processor and input/output attachments (1 of 3)

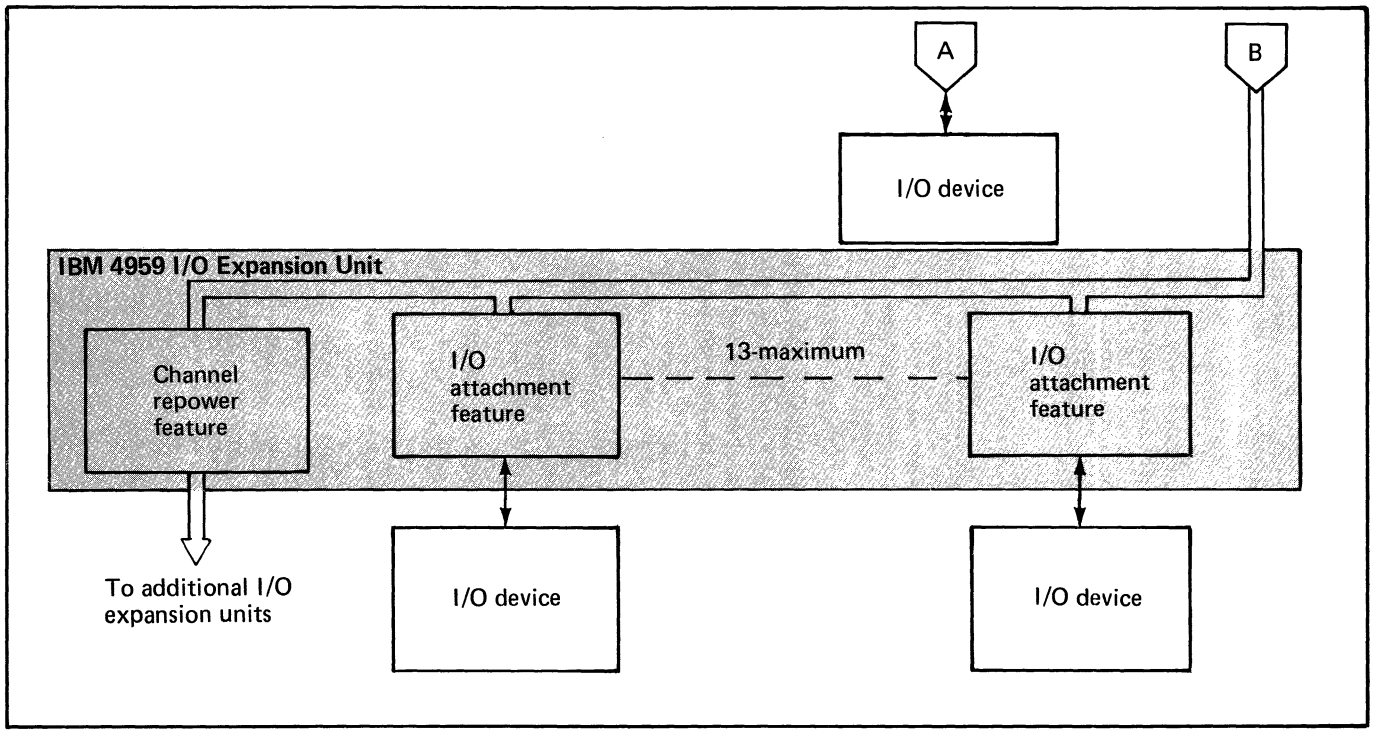


Figure 53. The Series/1 4955 Processor and input/output attachments (2 of 3)

All interfaces and attachments occur via printed circuit cards which plug into slots. These interfaces and attachments include:

- The optional relocation translator
- The floating-point processor
- The channel repower feature
- The two-channel switch

Figure 53. The Series/1 4955 Processor and input/output attachments (3 of 3)

Most of the interfaces between the input/output bus and the devices are microprocessor controlled as shown in Figure 54. Notice that the interface becomes specific to the attached device only beyond the microprocessor level. It is this commonality in interface design that integrates self-diagnosis, error detection, and error recovery into the hardware and software system. In the power-on state, the device may be logically disconnected from the input/output channel—under microprocessor control—and fully tested before being reconnected. Furthermore, the system checks the interface itself by passing signals back and forth across the bus prior to the startup of the application. The system attempts to isolate problems in this way—whether the device is IBM-supplied or user-supplied.

During the execution, the microprocessor performs checking in a device-dependent way—a capability allowed by the programability of the interface. Time outs, sequence checking, and parity checking are performed as appropriate. Errors detected are reported in standard form through the level status register (even, carry, and overflow) for use by either:

- The user's custom software
- The error recovery software of the Realtime Programming System
- The Event Driven Executive
- The Control Program Support package

Errors are also signaled by interrupts to the processor as explained later in this chapter.

The input/output bus itself is called an asynchronous multidropped channel. It is termed asynchronous because operations on the channel are always "hand-shaked" to assure correct transfer. That is, the system acknowledges each command or reply through an agreed-upon set of protocols. Such handshaking is actually contained in a set of timing and control signals passed back and forth on a subset of the bus lines as indicated in Figure 55. The channel is termed multidropped because all transfers are accompanied

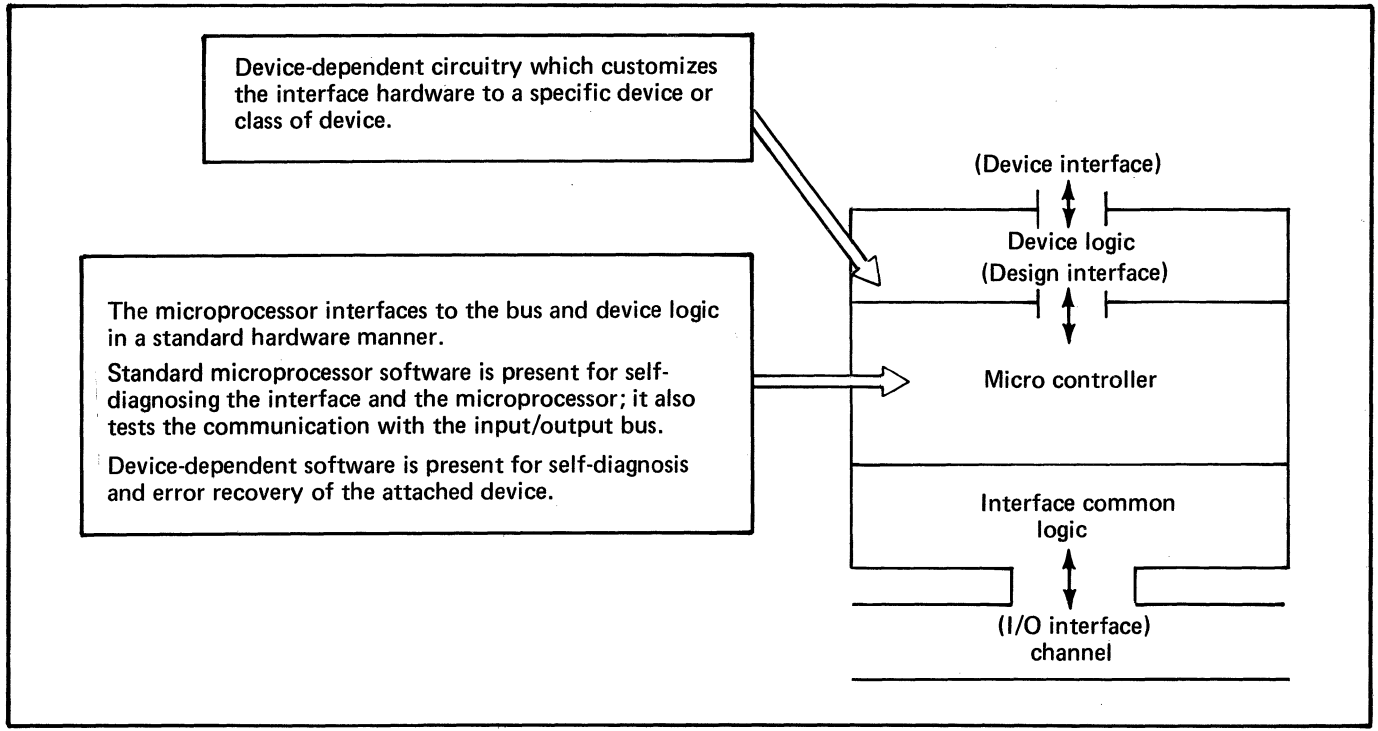


Figure 54. Organization of the microprocessor-controlled interface between the input/output channel and devices

by device addresses; all interfaces read these addresses, but only the addressed interface responds.

Transfer of data takes place on the bidirectional data bus. The device communicating with the processor is the one whose address (a number between 0 and 255) is placed on the address lines of the bus.

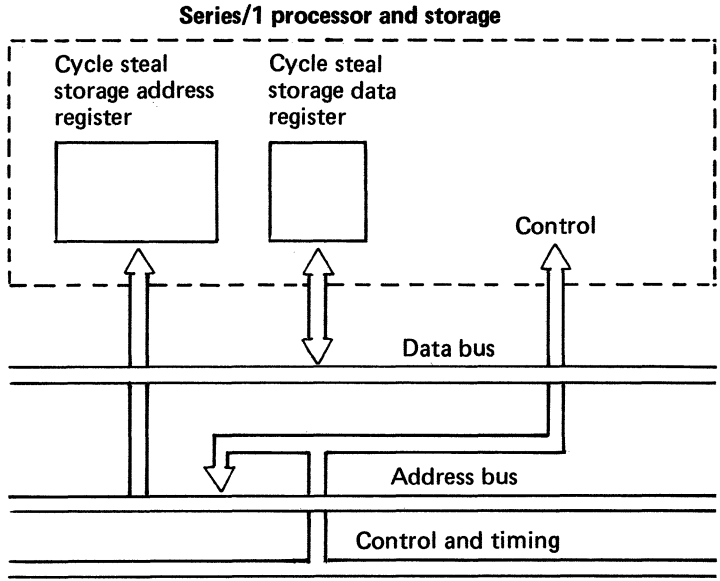
A transfer to a device involves passing an address and a command—together with data—in the appropriate direction. Handshakes and control signals pass on the control lines of the channel. High-speed devices transfer data directly into main storage without processor intervention. Cycle steal transfers must pass a main storage address along with the data. The 17-bit address bus permits full 16-bit main storage addresses to be passed. The 17th bit indicates either cycle stealing or a direct program control operation. For processors with main storage translation, this address is fully translated. This procedure is discussed further in this chapter under the section entitled "Input and Output in the Cycle Stealing Mode."

It is not appropriate to discuss here specific characteristics of the Series/1 input/output channel like signal levels and timing constraints. Users typically need knowledge of these characteristics when they design custom interfaces to most small computer systems. The Series/1, however, provides an OEM interface for this purpose; this interface presents a set of bus lines (simpler than the one provided by the input/output channel) to which users interface their devices. Details of this interfacing are covered in Chapter 9. By not interfacing directly to the input/output bus, system diagnostic capability is preserved.

Detailed input/output at the processor level involves control over device priority, interrupts from devices, and mode of transfer. Input/output at the processor level is discussed in the following two sections.

Input and Output Under Direct Program Control

Direct program control of input and output requires an explicit processor intervention in each data item transferred



Data bus

- 16 bits, plus two parity bits
- A bidirectional bus used to pass data and control information to and from devices
- During interrupt acceptance, device-address information and interrupt information is passed along this bus

Address bus

- 16-bit, bidirectional bus
- Used for device selection and passing commands to devices during input/output instructions

Figure 55. The Series/1 input/output bus: asynchronous and multidropped (1 of 2)

Control and timing

- During cycle steal operations, main storage addresses are passed along the address bus
- These addresses are fully mapped or translated in processors with hardware, storage address translation

Control lines

A set of unidirectional lines used for:

- Interrupt and cycle steal requests
- Condition code and status reporting
- Reset commands
- Basic timing and control of bus operations' sequences

Figure 55. The Series/1 input/output bus: asynchronous and multidropped (2 of 2)

across the channel. Under some conditions, it might be desirable to transfer a series of characters to or from a device. An example of this data transfer is depicted in Figure 56.

Polling vs. Interrupt-Driven Input/Output

There are two methods of software organization: polling, and interrupt-driven. Polling involves the repetitive testing of each device's status and the transmission of data when the device is available as shown in Figure 56 (2 of 3). This approach is *not* used in operating systems or applications—except in situations where special, very fast devices are present. Usually, it is much more efficient to use the interrupt-driven, input/output method of operation which involves the processor in the data transfer operation only when the device signals that it is ready to receive or transmit data.

Figure 56 (3 of 3) shows this process, indicating that the only time the processor is actually involved with the transfer of data to the device is when the device signals it is ready to receive a character. Normally, the processor is busy at some task level. The system periodically interrupts the processor to transfer a character, and then resumes execution of the interrupted task. In the Series/1, the interrupt occurs on a different priority level which has its own registers; consequently, there is little overhead expended in saving and restoring the state of the interrupted process. Direct program control operations take a very small portion of the processor's time for each input/output. Assuming moderate data rates, the interrupt-driven method is an efficient way to transfer data between devices. At fifteen characters per second, the overhead is negligible. However, devices transferring data at one thousand characters per second or higher begin to impact the throughput of the processor and increase the overhead of the operation.

A large number of devices with slow data transfer times will also use a significant amount of the processor's time. In these cases, the devices are more effectively connected in a cycle steal mode; such a connection decreases the processor's involvement significantly by putting much of the

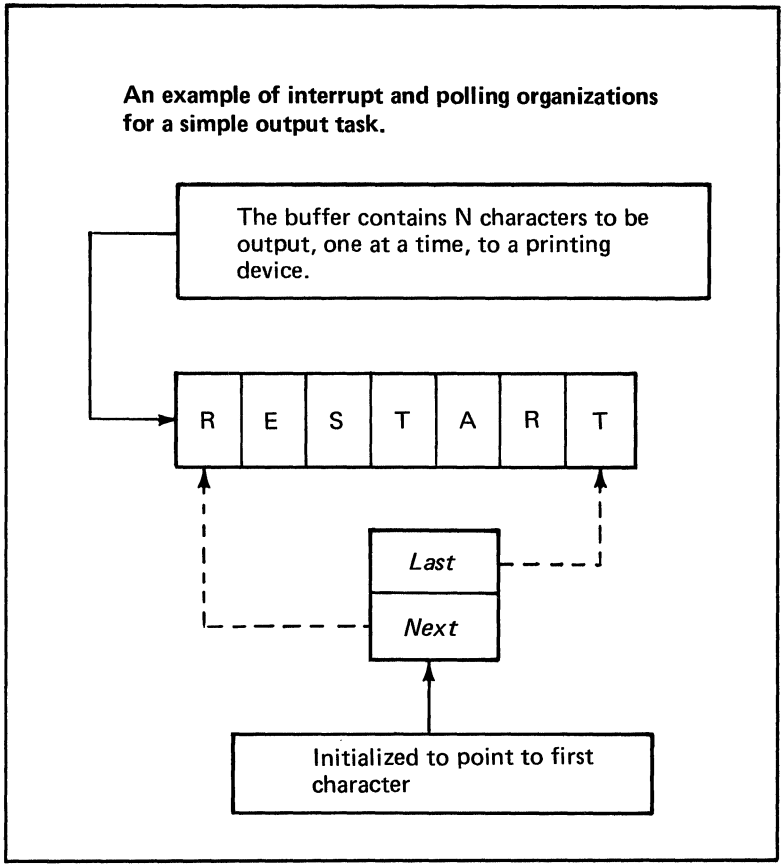


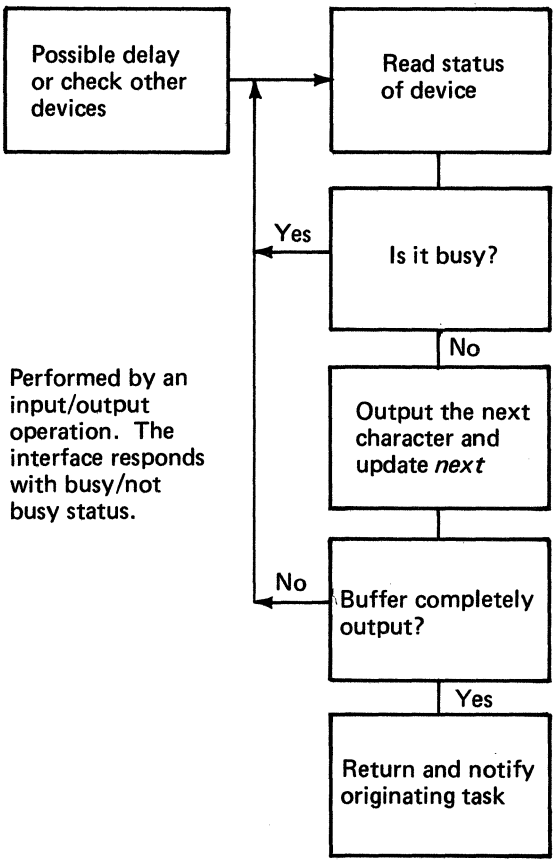
Figure 56. Direct program control of devices (1 of 3)

input/output control function into the interface. The Series/1 communications' interfaces are good examples of this method. Input and output in the cycle steal mode are discussed in the next section.

Effects of Buffering on Task Execution

Direct program control of a device affects the task initiating the input/output operation as indicated in Figures 57 and 58. A task which calls for a buffer to be filled from a specific device initiates a series of transfers. Each of these transfers takes only a short span of processor time to

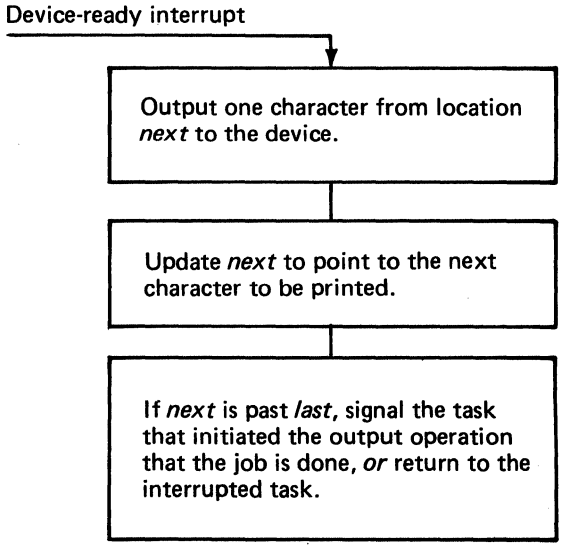
Polling input/output under program control

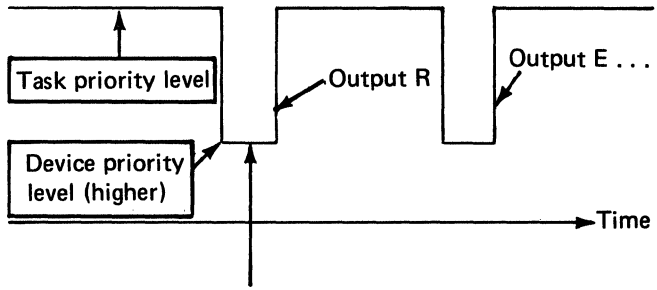


Because of the difference between the processor and the device speeds, this form of polling ties up the processor completely doing what is, essentially, a small task. By comparison, interrupt-driven input/output of this type operates with a very economical overhead.

Figure 56. Direct program control of devices (2 of 3)

Interrupt-driven input/output under program control





Overhead is less than half of one percent if:

- 50 microseconds are used to respond to the interrupt and to output one character, as above
- The device accepts 100 characters per second

Figure 56. Direct program control of devices (3 of 3)

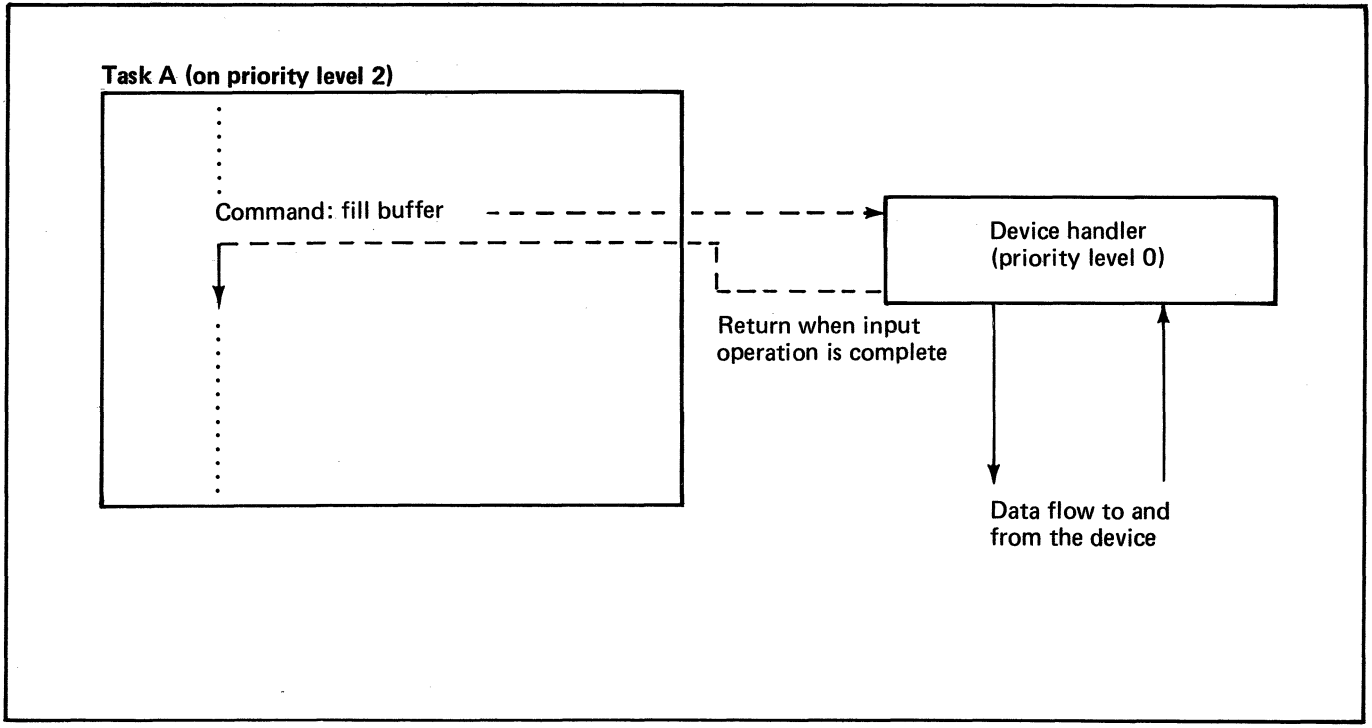


Figure 57. Effect of non-overlapped input/output on task execution (1 of 2)

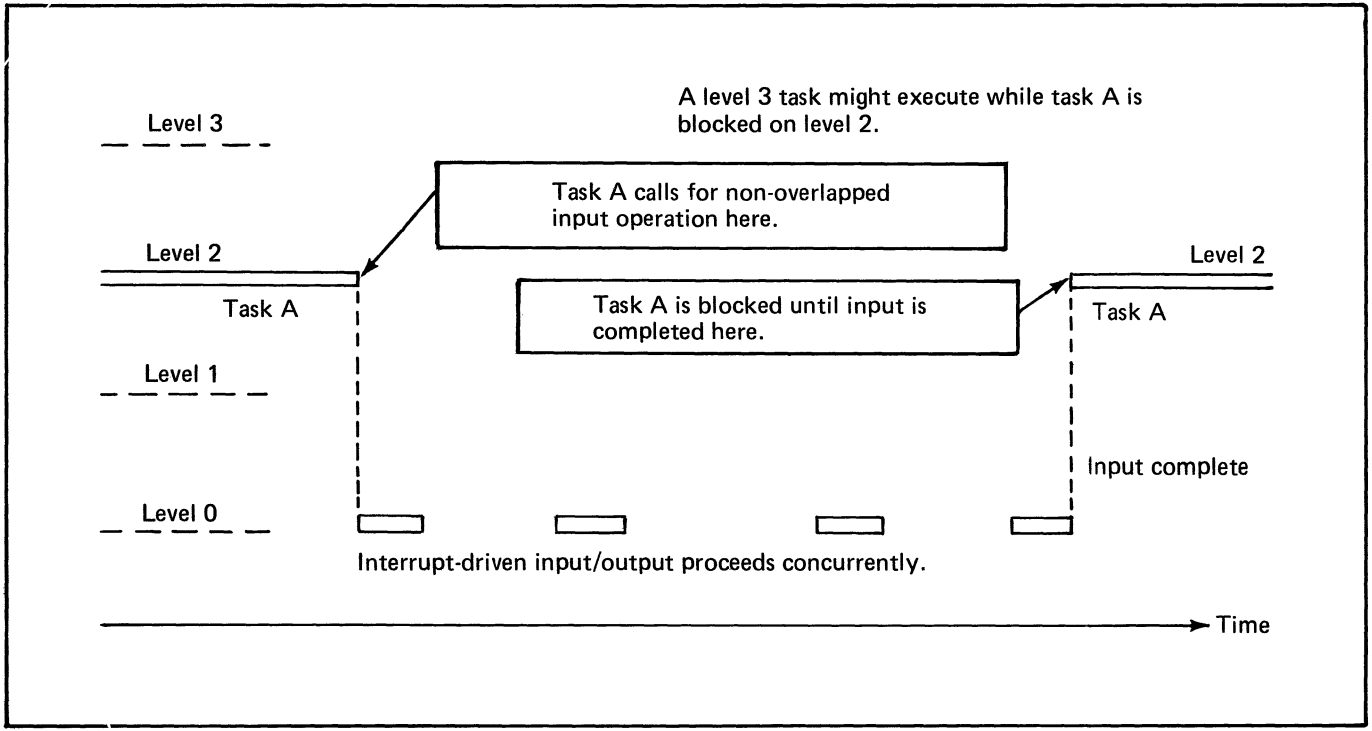


Figure 57. Effect of non-overlapped input/output on task execution (2 of 2)

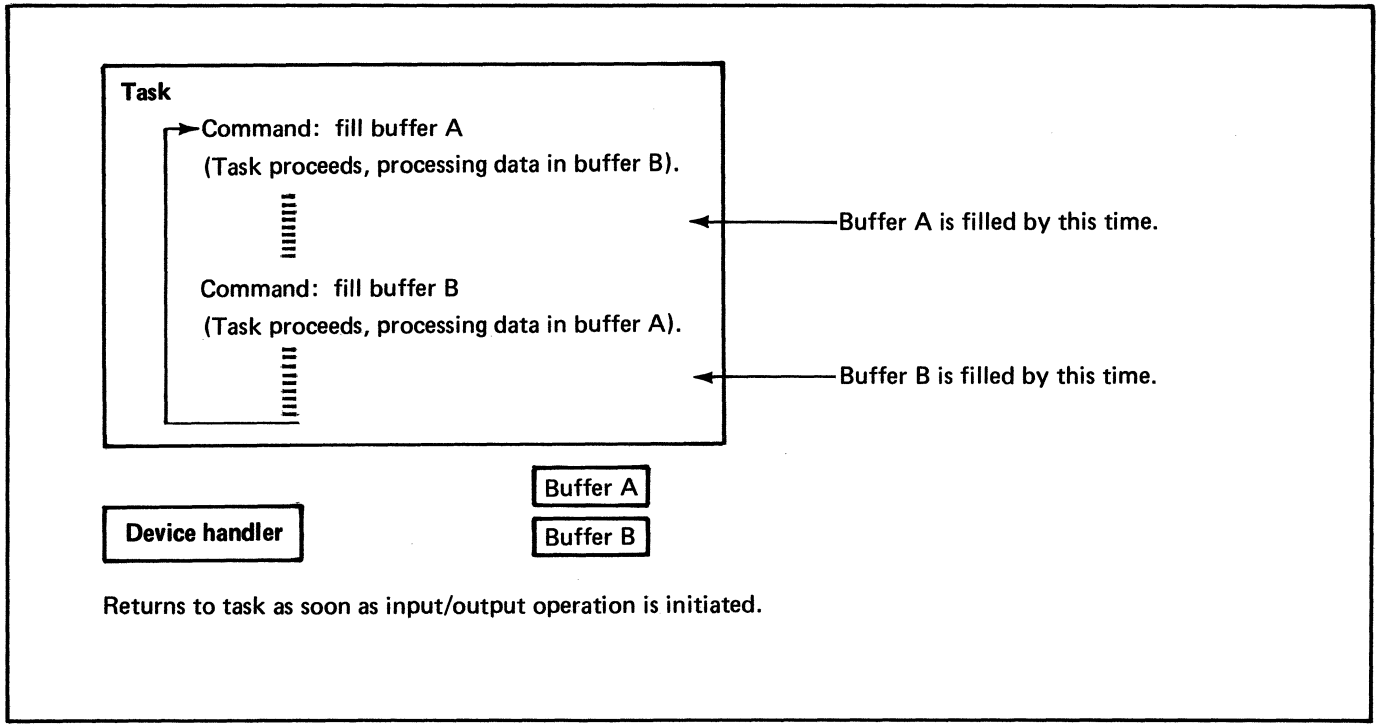
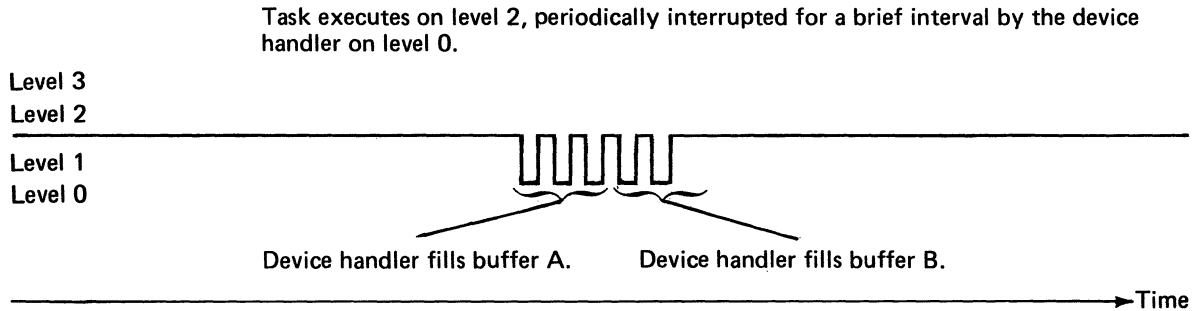


Figure 58. Direct program control and overlapped input/output (1 of 2)



A user can design a task with well-defined, input/output needs to execute concurrently with the input/output operation. In this example, two buffers perform that operation.

Before initiating the next task or processing the previous task, a task must determine that a requested, concurrent input operation is completed.

Series/1 software returns control to the initiating task immediately. The task then decides whether to continue execution—overlapped with input/output—or to block execution until the operation is complete.

Figure 58. Direct program control and overlapped input/output (2 of 2)

respond to the device interrupt (Figure 57). The processor may be busy in performing some other task in between data item transfers, but the task asking for the input operation may still be blocked throughout the duration of the data transfer. If the task needs the block of information before it can continue, it must wait for the completion of the data transfer. In this circumstance, overlapping the direct, program control input/output operations of one task with the execution of a different task (as in Figure 57) is an efficient way to use the processor. Sometimes the task initiating the input/output operation can actually proceed without waiting for the operation to complete (Figure 58). When that occurs a "double buffering" mechanism is deliberately set up by which the task initiates an input operation to fill one buffer while processing the other. The system, then, overlaps the input operation with the execution of the same task. Commonly, those input/output operations that arise in data acquisition and similar dedicated, small computer applications accomplish this double buffering. Direct program control is an efficient way to handle such situations; in critical applications, it also gives the programmer direct control over timing and response of the system. Software systems must permit the programmer to decide whether the processor should return to the task initiating the input/output operation immediately or only after completing the input/output. This programming option is discussed later in this chapter.

Direct Program Control Instructions

The actual involvement of the Series/1 processor in direct, program control input/output operations is straightforward: it involves only one instruction whose fields are coded with the specific operation desired. Figure 59 shows the Operate I/O instruction: a two-word instruction whose effective address points to a two-word package, the immediate device control block (IDCB). The immediate device control block, in turn, contains all the information specific to an input/output operation including the device address (eight bits indicating one of 256 devices), and a command. The full-word, immediate

field within the device control block is the source of data for an output operation and the destination of data for an input operation. For a transfer involving a byte instead of a full word—the length of the data transfer depends upon the particular device being addressed—the transfer is to and from the last significant half of the immediate field.

Notice in Figure 59 how the input/output channel transfers the command and device address along the channel address bus for recognition and interpretation by the appropriate device interface. The control information is not depicted.

Various commands are necessary for complete device control, especially when special devices are interfaced to the processor. Figure 60 lists these commands. The command portion of the immediate device control block is eight-bits wide: four bits identify the generic command; four bits are available as sub-commands. For example, in addition to the expected read and write commands, there exists a read identification (Read ID) command. This command causes the addressed device to return a sixteen-bit identification word to the immediate field of the immediate device control block. The identification word contains:

- A unique identification code for the device so the processor can determine physical characteristics and input/output requirements for the device
- An indication of whether the device runs under direct program control or cycle steal mode
- An indication of whether the device is a standard IBM device or an OEM device
- An indication of whether the device is controller-interfacing several devices or not

This information could, of course, be programmed into the system rather than being available under software control. However, making this information available to programs insures that:

- Error checking can be done
- Error recovery and startup can be expedited

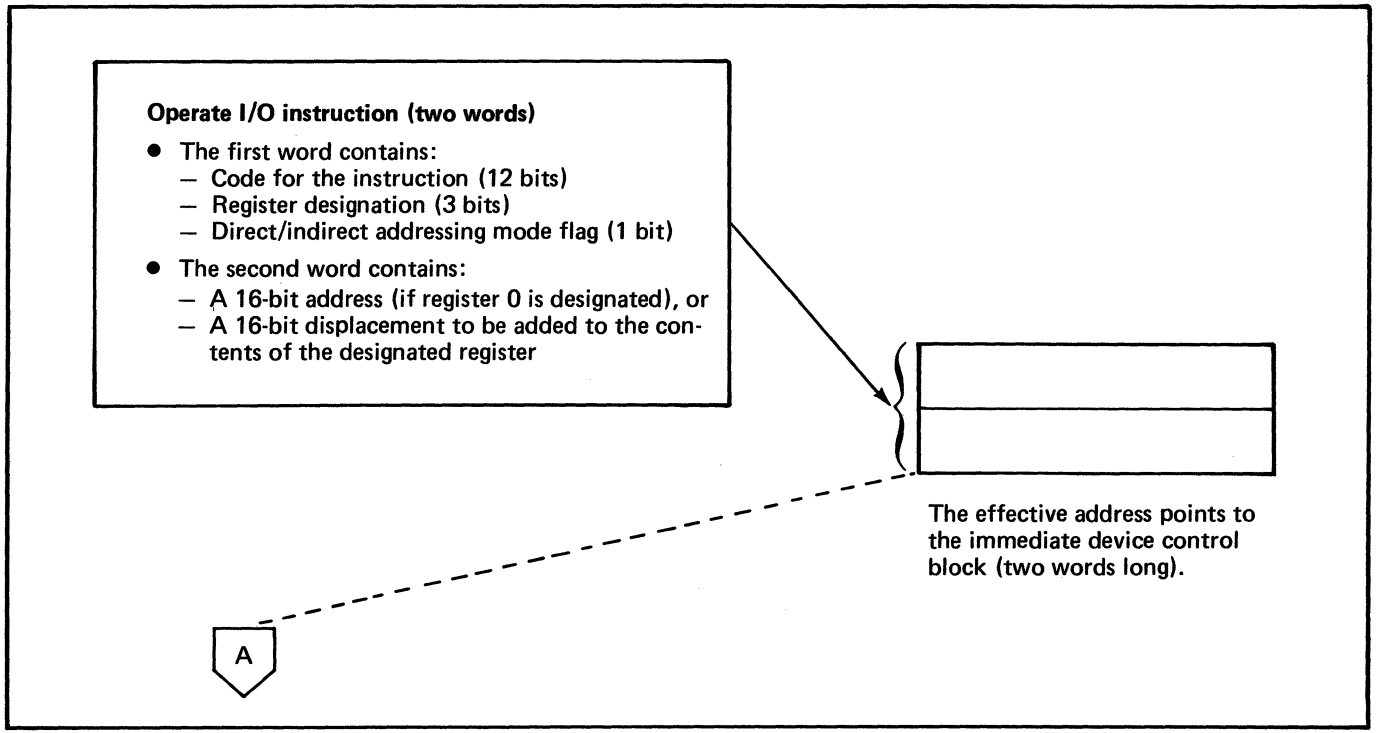


Figure 59. Direct program control performed with a single instruction—Operate I/O (1 of 3)

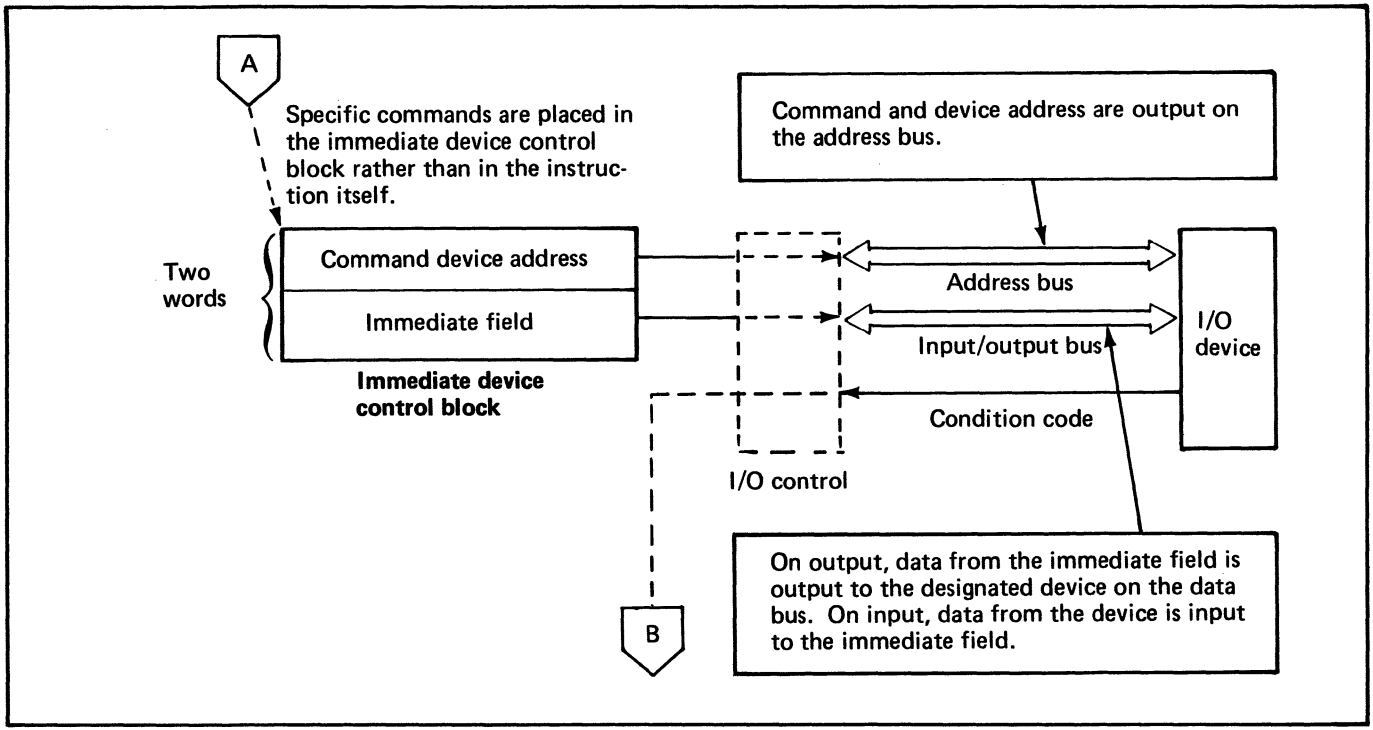


Figure 59. Direct program control performed with a single instruction—Operate I/O (2 of 3)

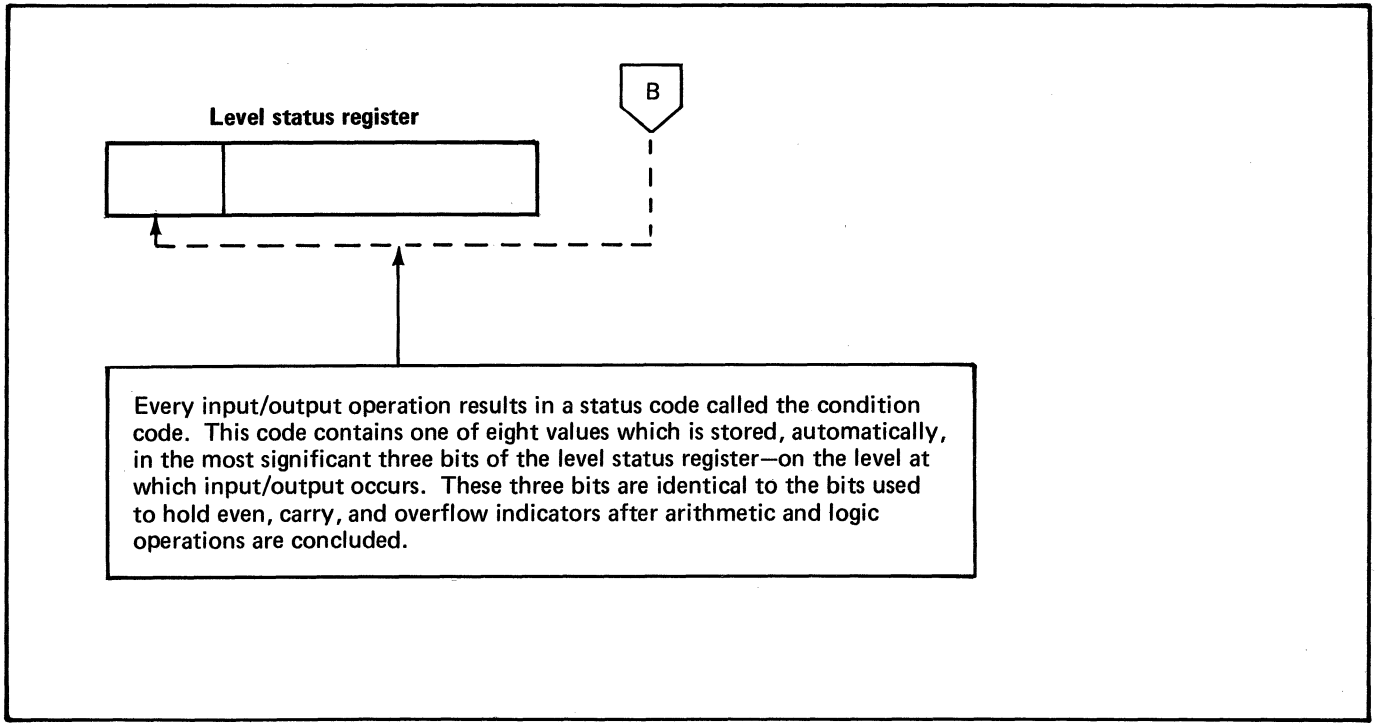
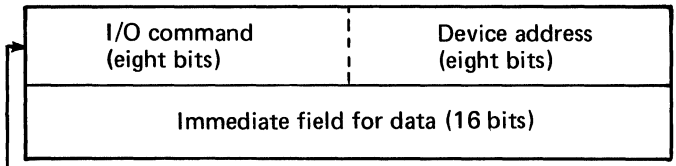


Figure 59. Direct program control performed with a single instruction—Operate I/O (3 of 3)

Immediate device control block (two words)



Direct program control commands

- Read—Transfers a word of data or a byte of data from the addressed device into the immediate field of the immediate device control block. If a byte transfers, it occupies the least significant byte of the immediate field, and the other byte is zeroed.
- Write—Transfers a word or byte of data from the immediate field of the immediate device control block to the addressed device. If a byte transfers, it is the least significant byte of the immediate field.
- Read ID—Transfers an identification word from the addressed device to the immediate field of the device control block. The device identification word contains information about the device and may be used—either during startup or as part of error checking—to determine the devices that are attached to the system.
- Read Status—Transfers a word of current device status information from the addressed device to the immediate field of the device control block. Interpretation of the bits and fields depends upon the specific device.
- Prepare—Transfers a word to the addressed device from the immediate field of the immediate device control block. The transferred word controls the device's interrupt level and enabling flag (see Figure 61).
- Device Reset—Resets the addressed device including clear, any pending interrupt, or busy condition

Figure 60. The major input/output commands for direct program control of devices

- Proper operation of devices can be performed from common, system software

For example, knowledge of whether or not a device is IBM-supplied determines whether or not certain levels of self-diagnosis will be performed.

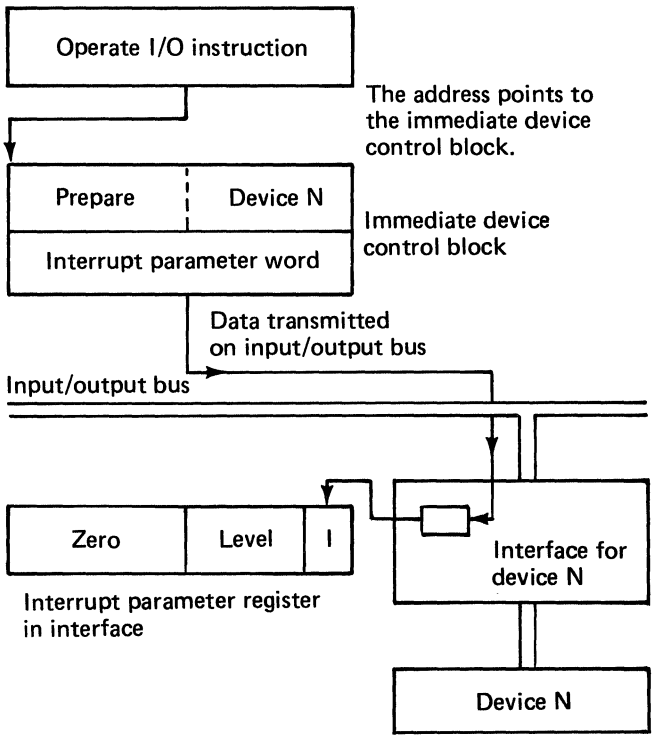
The Read Status command transfers a word from the device interface to the immediate field of the immediate device control block. The same process occurs during a Read or Read Identification (Read ID) command but in the latter case, the data codes the status of the particular device. This coding might include a busy indication, error detection information, or a power status indication depending upon the device. Similar commands to the interface from the immediate field of the immediate device are: Control, Reset, and Prepare.

Control passes a word from the device control block immediate field to the device interface. What the interface does with the word depends upon the device itself. Among the actions initiated are the following:

- To abort an operation in process
- To send a device to some standard state
- To position an electromechanical system
- Any other operation:
  - Which must be treated differently from data transfers, and
  - Which the interface designer builds into the system

The availability of the Control command is clearly important to OEM device interface designers. Reset is a special kind of control which also resets the common, system state including any pending interrupt.

The Prepare command controls the interrupt level of a device and the basis on which a device generates interrupts (Figure 61). As stated earlier in this chapter, interrupts may be masked on a system wide basis, on a level basis, or on an individual device basis. The hardware priority level for a device is not determined by its location in the processor or input/output expansion units. This determination is made by information stored within the device interface itself and



Level: A four-bit area designating on which of the 16 priority levels the device should respond. Although Series/1 architecture supports up to 16 levels, only four levels are recognized on current processors.

I: Device mask. When set to 1, the device interrupt is enabled; when set to 0, disabled. These settings provide:

- Control over specific, individual device interrupts (device mask)
- Control over individual levels of priority interrupt (level mask)
- Control over all interrupts on all levels (summary mask)

Individual devices may have their priority level set under program control, and be individually enabled or inhibited under program control.

Figure 61. Individual devices under program control

subject to change under program control. The Prepare command transfers a single word to the device. The transferred word contains the levels of priority on which the device acts (note that the Series/1 architecture allows up to 16 levels of priority but that currently available processors recognize only four levels). One bit is used to indicate whether the device is permitted to interrupt or not, in exactly the same fashion as the determination is made by the interrupt control information built into each level of the processor. IBM has designed the interfaces to accept the Prepare command at any time; this enables a user to change the level and interrupt status of any device at any time under program control.

Normally the system software, like the Realtime Programming System, carries out all control over interrupt level, interrupt inhibiting, and similar functions. In custom-tailored applications, however, users might assume this control on their own. The flexibility and generality of the direct, program control commands give the user the power to run special devices in whatever way the application dictates.

Error Detection and Reporting

This book has repeatedly emphasized that robust system design requires good error detection. Without this detection, thorough error recovery cannot be built into the software system. For this purpose, each input/output operation inputs status information to the processor. A condition code is a three-bit code used to indicate the result of an input/output instruction or the reason for an interrupt (Figure 62). The three-bit condition code is stored within the level status register in the three bits used for even, carry, and overflow information. There is no conflict in dual use of the level status register because:

1. Interrupts and input/output are normally performed on priority levels different from those used by tasks
2. Registers are duplicated on each priority level

Depending upon the actions of the device, interface, and hardware, the condition code is set as indicated in Figure 62;

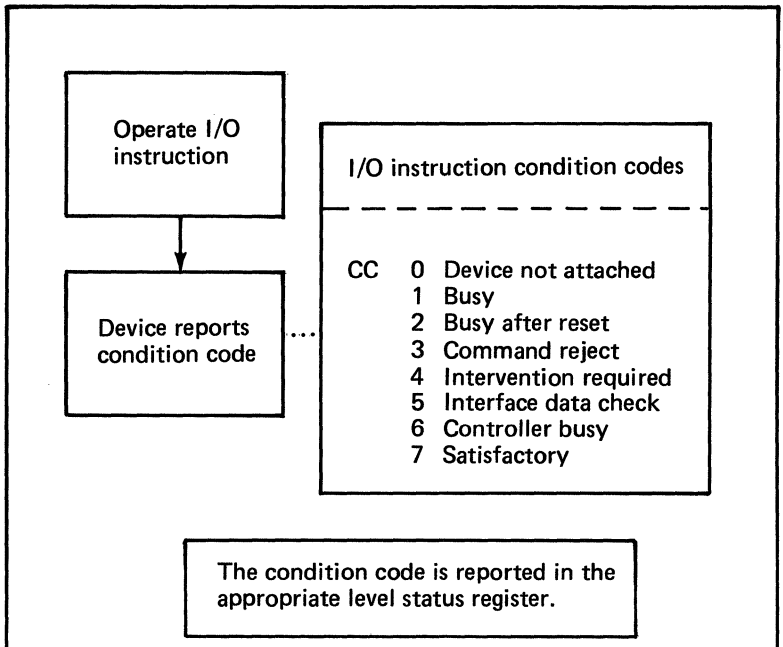
a variety of conditional branching instructions can then interrogate the code. With this system design, software can:

- Perform an operation
- Check to determine whether or not the operation was carried out satisfactorily
- Determine the source of difficulty if the operation was not satisfactory
- Handle all special cases

Overall Operation of Direct Program Control Input/Output

Overall operation of direct program control, input/output transfers involves several steps. First the system prepares the device, setting the interrupt priority level and interrupt mask with the Prepare command. Secondly, each time the device is ready to input or accept data, an interrupt is generated (subject, of course, to masks and current processor level control). As shown in Figure 63, the interrupt then causes a condition code to be placed in the level status register on the level of the device. As explained in the discussion of interrupts, an interrupt identification word is input to register seven on the device priority level. The least significant byte of register seven contains the device address which the interrupt hardware uses to branch to the interrupt response routine. The other byte contains either device-dependent information needed by the responding software routine or more detailed error information. Figure 64 defines the interrupt identification word and the additional error information detected by the interrupt.

Assuming no error occurred, the processor uses the device address in register seven to access a table which contains the addresses of blocks of information related to the devices. In IBM-supplied software, this block is called a device descriptor block (DDB) and includes many parameters related to error recovery, buffers, and storage locations. The first word of the device descriptor block is the address of the interrupt response routine itself; it is this address that the system fetches and uses as the starting address for program execution.



CC=0 *Device not attached.* Reported by the channel when the addressed device is not attached to the system.

CC=1 *Busy.* Reported by the device when it is unable to execute a command because it is in the busy state. The device enters the busy state upon acceptance of a command that requires an interrupt for termination. It exits the busy state when the processor accepts the interrupt. Certain devices also enter the busy state when an external event occurs that results in an interrupt. When this condition code is reported, a subsequent priority interrupt from the addressed device always occurs.

CC=2 *Busy after reset.* Reported by the device when it is unable to execute a command because of a reset and the device has not had sufficient time to return to the quiescent state. No interrupt occurs to indicate termination of this condition.

Figure 62. Definition of the eight condition codes which may be reported after each input/output instruction (1 of 2)

CC=3 *Command reject.* Reported by the device or channel when:

1. A command is issued that is outside the device command set
2. The device is in an improper state to execute the command
3. An incorrect parameter was supplied to the I/O command

CC=4 *Intervention required.* Reported by the device when it is unable to execute a command due to a condition requiring manual intervention to correct.

CC=5 *Interface data check.* Reported by the device or the channel when a parity error is detected on the I/O data bus during a data transfer.

CC=6 *Controller busy.* This condition is reported by a device controller, not the addressed device, when the controller is busy. It is reported only by controllers that have two or more devices attached (each device having a unique address). When this condition code is reported, a subsequent controller-end interrupt always occurs.

CC=7 *Satisfactory.* Reported by the device or channel when it accepts the command.

Condition codes reported after an Operate I/O instruction are different from the condition codes reported after an input/output interrupt.

Figure 62. Definition of the eight condition codes which may be reported after each input/output instruction (2 of 2)

Series/1 hardware performs all of these operations, minimizing the overhead expended in responding to interrupts. It is the interrupt response routine which actually performs the direct program control input/output operations, error-checks the results, and sets up for the next transfer. Software handles most of the complex input/output activity. The logical and physical buffer sizes described here are only

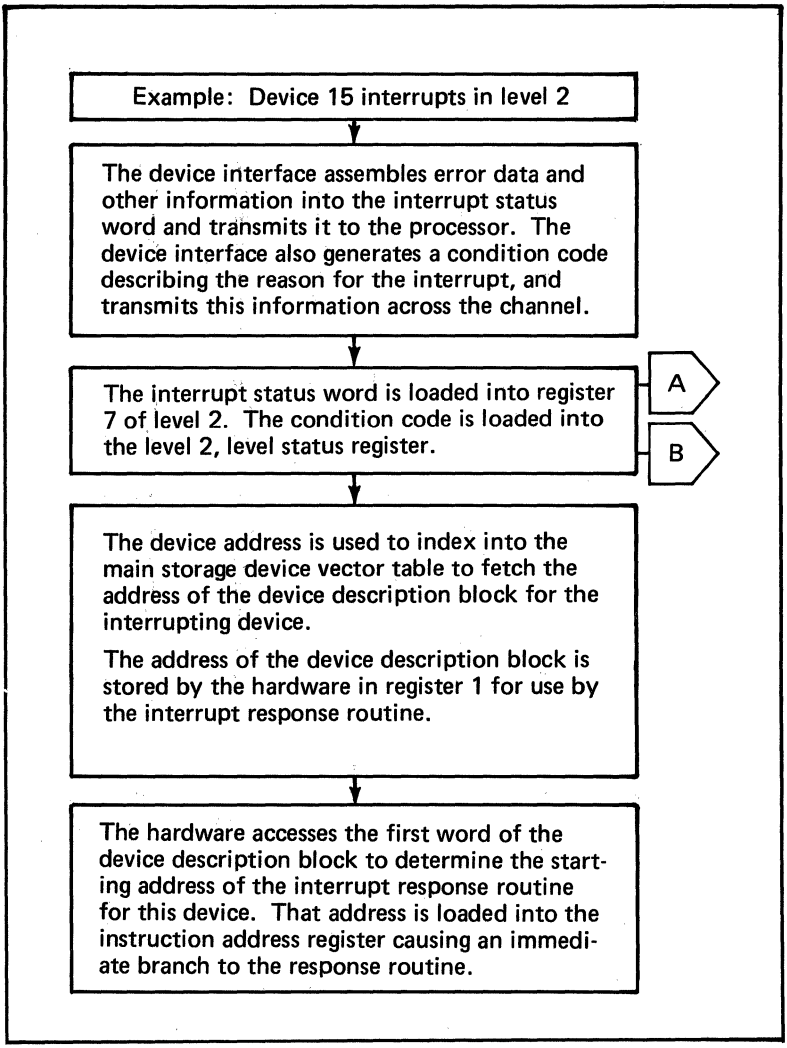


Figure 63. Condition codes accompanying each input/output interrupt (1 of 5)

simple examples of additional information needed at the higher levels. This book, in a later section, briefly discusses software for control of input/output at the higher levels. The more complex examples are covered there.

Of special interest to small computer applications is the structure of the direct program control command in Figure 56. Because the immediate device control block

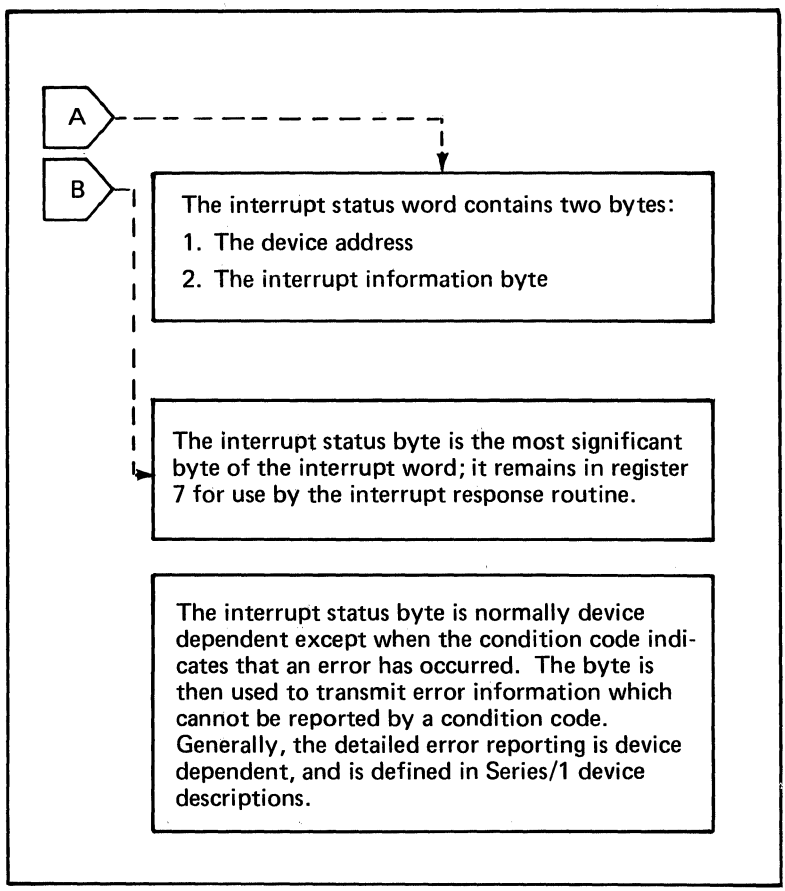


Figure 63. Condition codes accompanying each input/output interrupt (2 of 5)

is separate from the Operate I/O command, it is possible to write input/output software which can control multiple, identical or similar devices in a very efficient manner. Figure 64 shows a common input/output control routine which addresses different immediate device control blocks (perhaps, depending upon a number in a register) each time it executes. Because the control-word device addresses and immediate data fields are separate from the input/output command, the same code can control multiple devices without interference. Similarly, input/output commands—which prepare devices for operation—address different control blocks with different information.

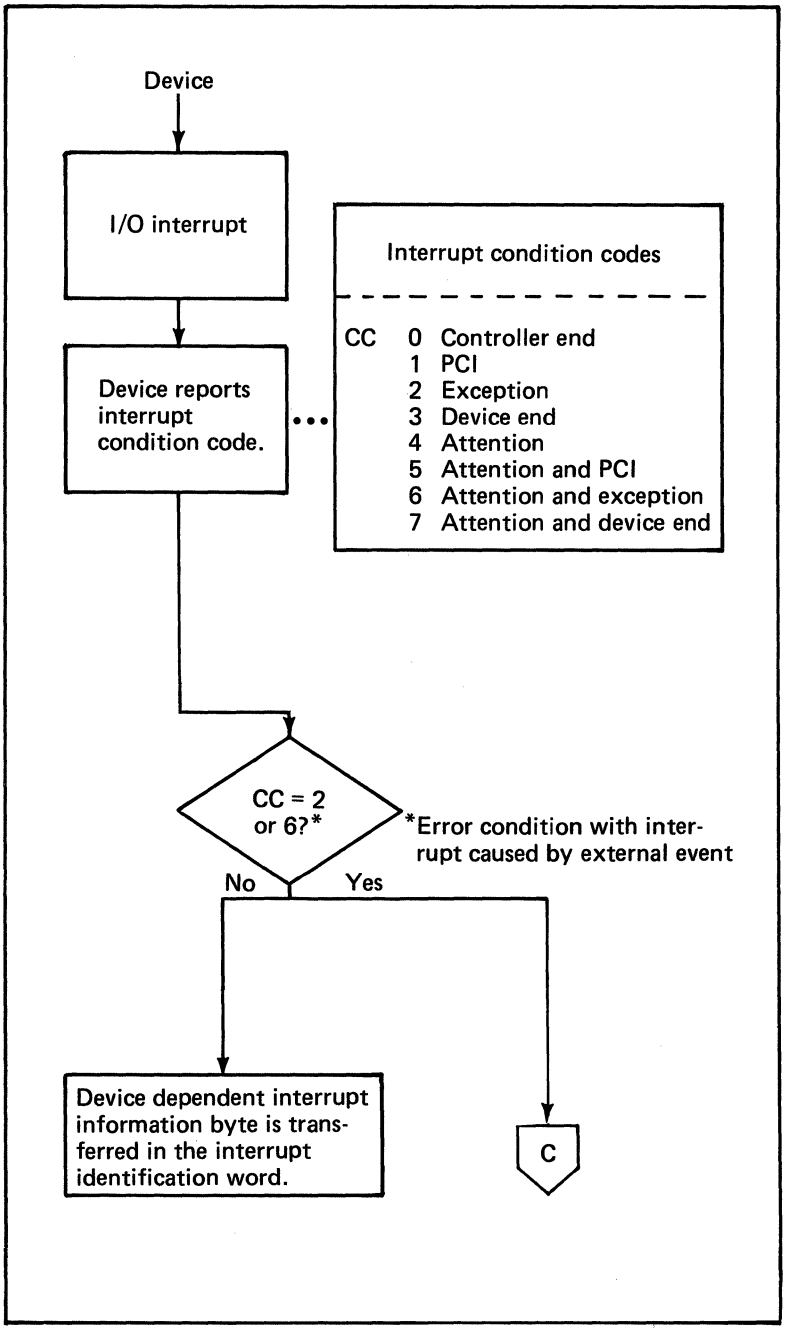
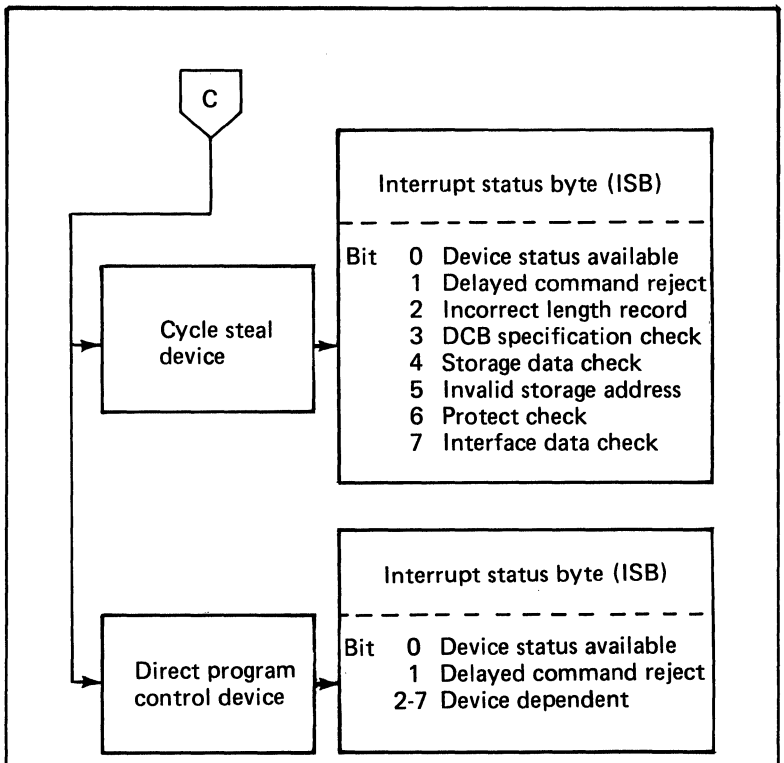


Figure 63. Condition codes accompanying each input/output interrupt (3 of 5)



CC=0 *Controller end.* Reported by a controller when controller busy (I/O instruction condition code) has been previously reported one or more times. It signifies that the controller is now free to accept I/O commands for devices under its control. The device address reported with controller end is always the lowest address (numerical value) of the group of devices serviced by the controller. The interrupt information byte, in the interrupt ID word, is set to zero.

CC=1 *Program controlled interrupt.* Reported when the interrupt indicates that a DCB with the PCI bit set to one has been transferred by cycle steal to the device and no error or exception condition has occurred.

Figure 63. Condition codes accompanying each input/output interrupt (4 of 5)

CC=2 *Exception.* Reported when an error or exception condition is associated with the interrupt. The condition is described in the interrupt status byte (ISB) or in device dependent status words.

CC=3 *Device end.* Reported when no error, exception, or attention condition has occurred during the I/O operation, and the interrupt is not the result of a PCI. For example: an operation has terminated normally.

CC=4 *Attention.* Reported when the interrupt was caused by an external event rather than execution of an Operate I/O instruction. Additional status information is not provided unless the event requires further definition: for example, code bits for a keyboard function.

CC=5 *Attention and PCI.* Reported when attention and PCI are both present.

CC=6 *Attention and exception.* Reported when attention and exception are both present.

CC=7 *Attention and device end.* Reported when attention and device end are both present.

The advantages of the microprocessor-controlled interfaces and channels are apparent here:

- The hardware carries out a complex series of operations to facilitate low overhead and rapid event response
- The hardware error-checks every operation
- The hardware provides extensive diagnostics in an integrated, consistent manner

Each input/output code is accompanied by a condition code reported in the level status register. The condition code defines the reason for the interrupt and may be tested to guide the interrupt response routine.

Figure 63. Condition codes accompanying each input/output interrupt (5 of 5)

In fact, the common code might address a device with a Read Identification command and use the results to determine which device control block to use in the remainder of the code; this procedure would allow one routine to control multiple, dissimilar devices.

Input and Output in the Cycle Stealing Mode

Cycle stealing, in contrast to direct program control, involves much less explicit interaction on the part of the processor during the transfer of a block of information between main storage and a device. This has been accomplished in the Series/1 by building a great deal of processing capability in the input/output channel controller. As a result, the channel controller can handle many of the functions which the processor handles during a direct, program-controlled operation. Figure 65 illustrates this procedure—the processor executes a task whose code is stored in main storage. Execution involves many things, including:

- Fetching instructions from storage
- Fetching data items
- Performing computation using the arithmetic and logic unit
- Putting results back into storage

In other words, the processor uses the storage repeatedly. Each read or write storage action takes a length of time called a memory cycle; during this time nothing else can use the main storage.

Simultaneous with the execution of a task, the input/output channel cycle steals data between main storage and a device. Each such transfer involves many of the same physical operations discussed in the introduction to this chapter: addresses and commands on the bus, handshaking, error checking, and data transfer. However, during a cycle steal data transfer, the channel controller is responsible for all these functions. The only interaction with the processor is through contention for main storage. Each time a data item has been obtained from a device during an input

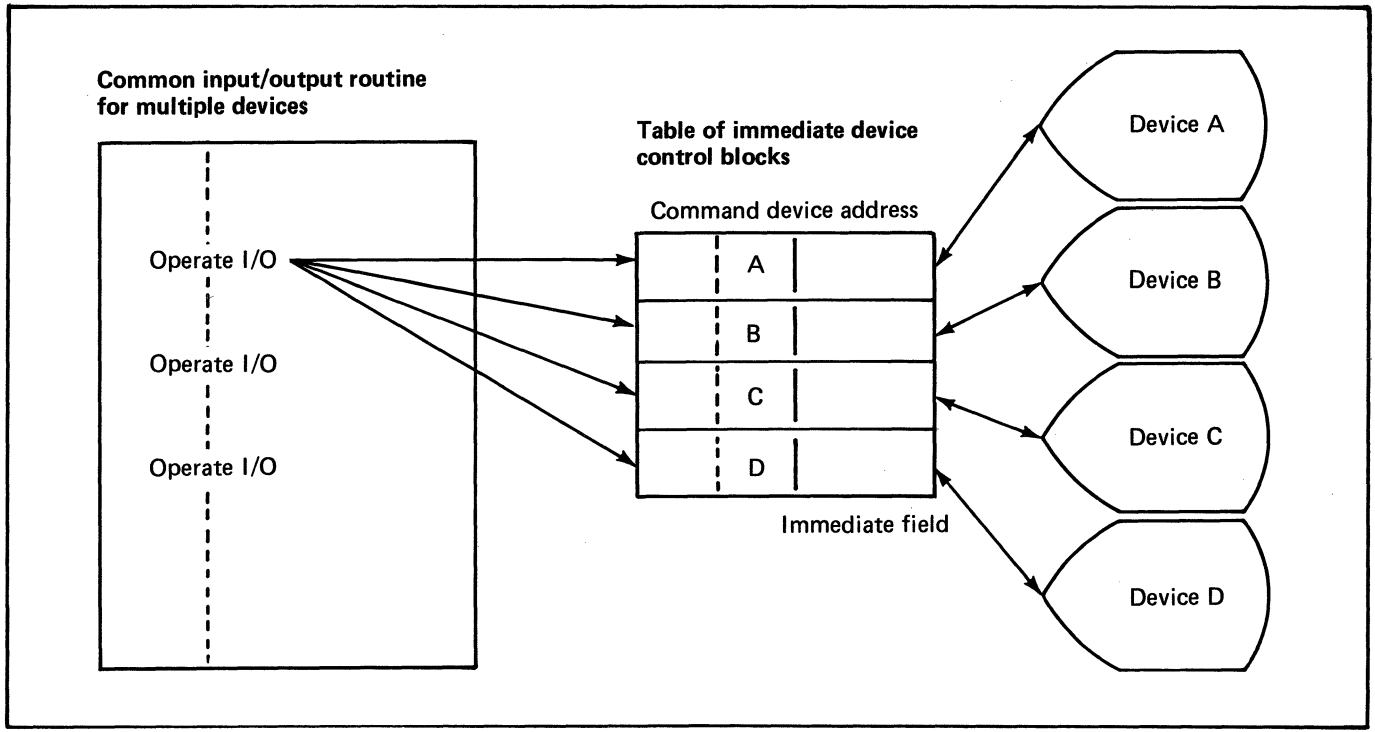


Figure 64. A common input/output control routine addressing different immediate device control blocks (1 of 2)

Each call on the input/output routine indexes to access the appropriate immediate device control block.

Separating the device address, the command, and the data area from the Operate I/O instruction itself creates practical input/output routines. These routines control multiple devices of different types and decrease the size of the overall system software.

Figure 64. A common input/output control routine addressing different immediate device control blocks (2 of 2)

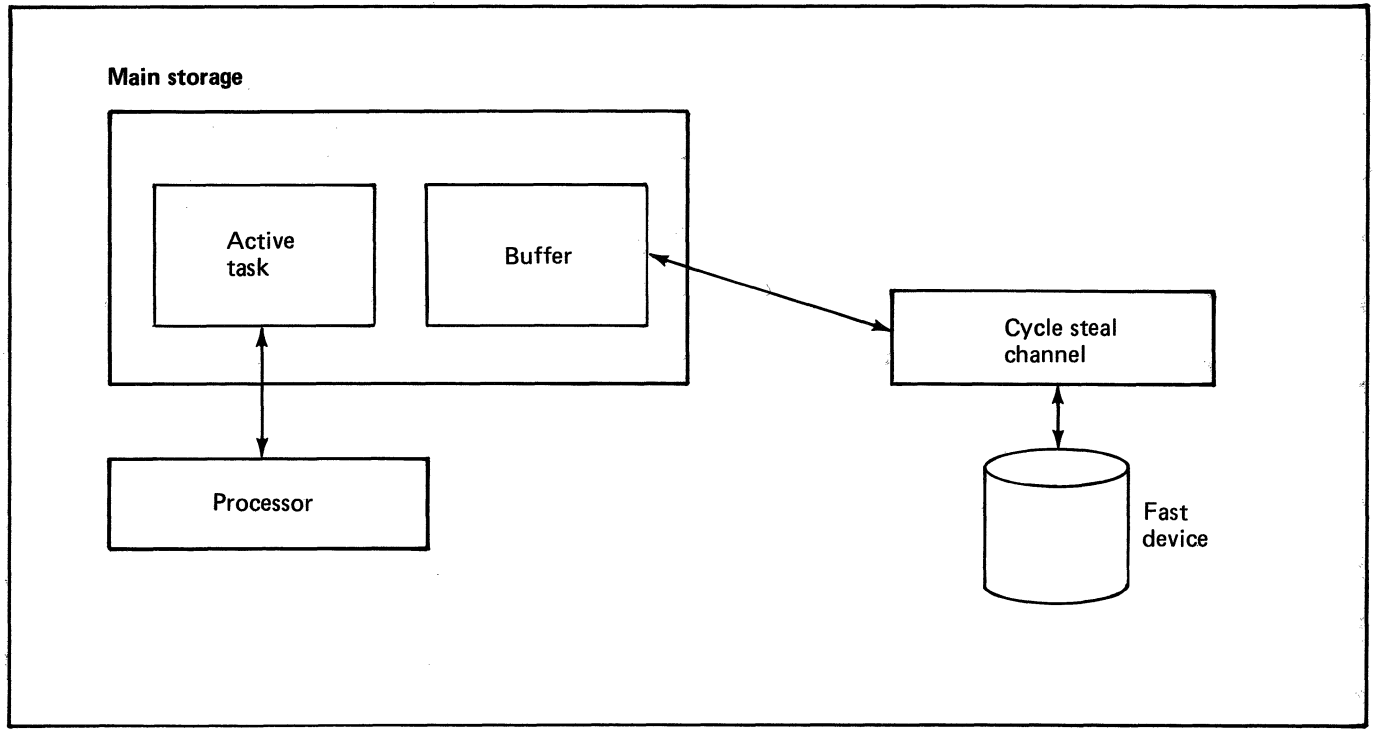


Figure 65. Cycle stealing input/output (part 1) (1 of 2)

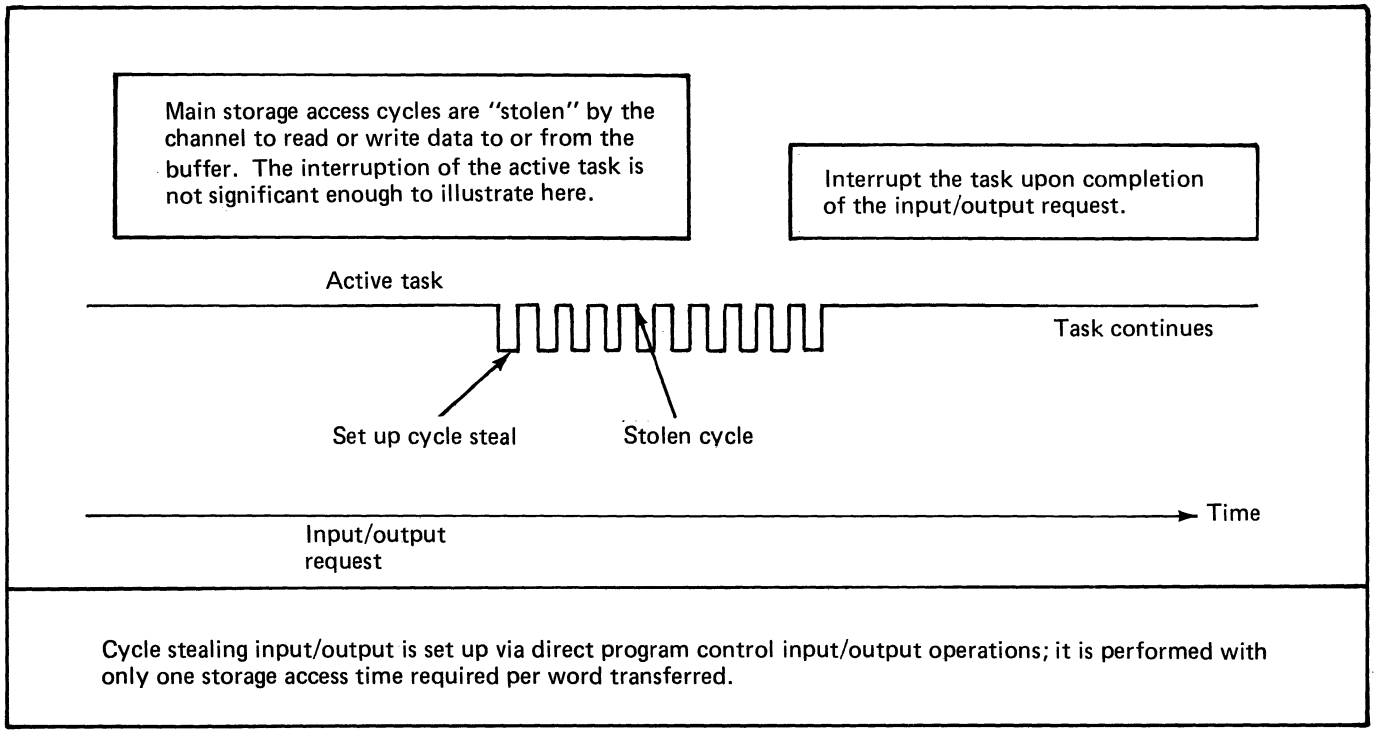


Figure 65. Cycle stealing input/output (part 1) (2 of 2)

operation, the channel controller must present the data—together with a main storage address—to main storage in exactly the same way the processor presents an address and data when it wishes to write in main storage. As indicated during the discussion of the processor in Chapter 3, both the processor and the channel have registers for accessing storage. A storage controller acts as a traffic policeman or priority arbitrator when both the processor and controller simultaneously contend for storage. Because the channel inputs data periodically and because each such data transfer has exclusive access to the main storage for one cycle during the transfer, the processor periodically finds the main storage busy and must wait—hence, the name cycle stealing for this mode of data transfer.

Once initiated, cycle stealing entails little overhead on the part of the processor. Consequently, cycle stealing is an economic process for faster devices whose servicing under direct program control would considerably decrease the throughput of the system.

Use of Microprocessors in Cycle Steal Controllers

The Series/1 technology uses a microprocessor—whose power resembles that of a small computer—in most device interfaces. In the current iterations of the Series/1, it is economically attractive to use cycle stealing interfaces even for relatively low-speed, input/output devices. One significant advantage of this technology is that the microprocessor can be programmed to perform the basic input/output operation as well as error checking, error recovery, self-diagnosing, and those special functions pertinent to a particular device. Communications-device data management in the Series/1 is an excellent example of this advantage. The interface can both transfer characters from a terminal-type device to main storage, and also check each character to see if it is a special control character requiring a special response from the processor.

An example of this ability occurs when a character signals the user who wants to backspace and overwrite something already typed on the terminal. The user wants an immediate

response to this command; the system must make the correction before a complete line is input. The programmed communications' interface could perform this backspace function or, alternatively, allow the processor to respond by interrupting it when the interface detects the backspace character. As a result, the processor is not engaged for most characters except when they occur at the end of an input line. In a communications' and terminal oriented application, this method of operation off-loads the processor very significantly and is a good example of the integrated hardware, software, and system design prevalent in the Series/1.

Cycle Steal Input/Output Instructions and Commands

The user must supply multiple parameters in the input/output command because cycle stealing operations transfer more than one data item without processor intervention. Figure 66 shows the form of the cycle steal, input/output command; it is identical to the command used in direct program control except for the interpretation of the immediate data field. As with direct program control, the system uses the same Operate I/O command with an address pointing to an immediate device control block. The command field, however, is specific for cycle stealing—Start I/O. This command field is a signal to the channel that the immediate field is actually the address of another block of information—the device control block—which contains the parameters necessary to carry out the transfer. These parameters include:

- An address in main storage where a data buffer resides
- A count of the number of bytes to be transferred
- A control word which is, essentially, a command to its cycle stealing controller

The command is part of the device control block rather than the immediate device control block because the system may often perform long sequences of cycle stealing operations without processor intervention. The system can accomplish this operation by having the following:

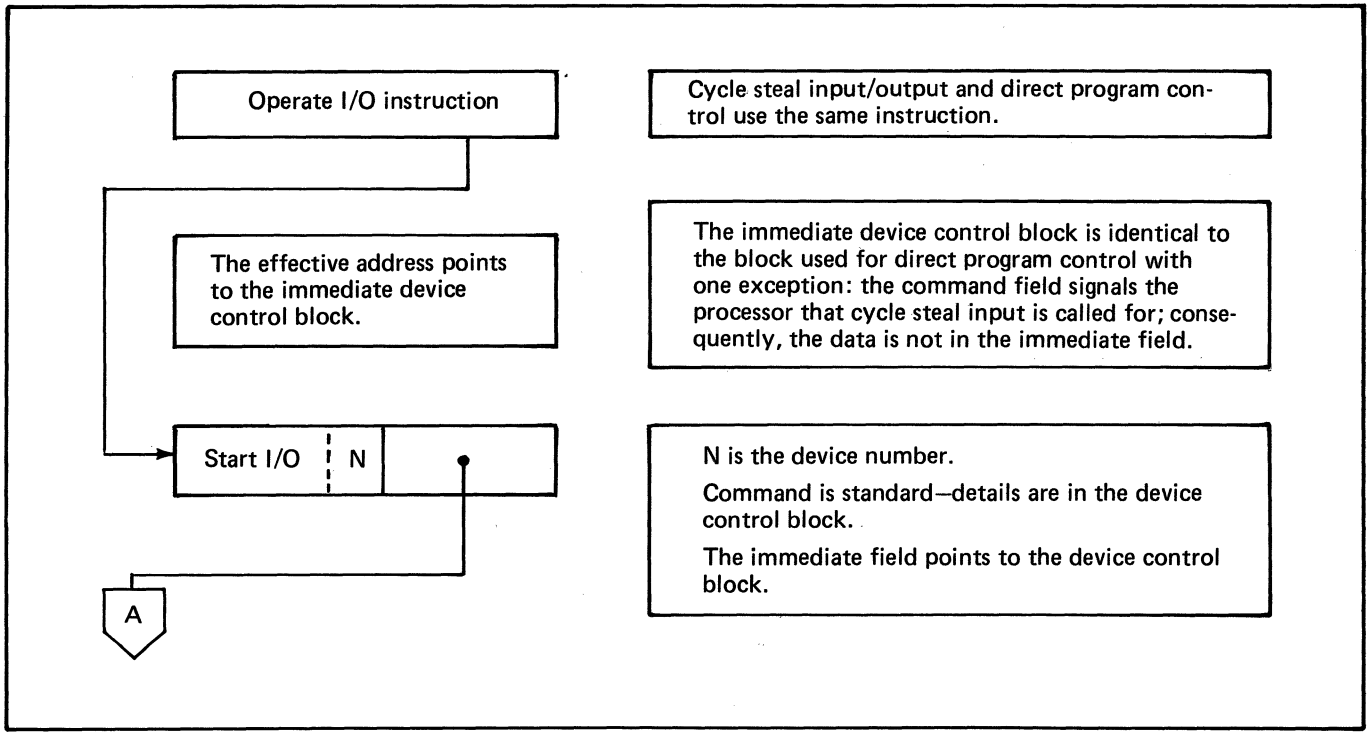
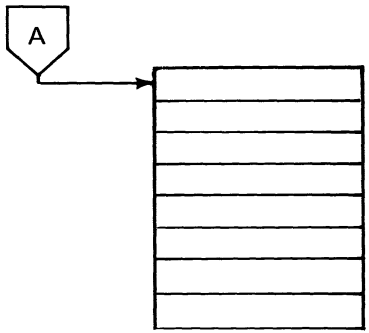


Figure 66. Cycle stealing input/output (part 2) (1 of 2)



Cycle steal input/output is controlled through an eight word data structure.

Direct program control input/output instructions set up the cycle steal input/output which the Start I/O command initiates. The command has a format similar to the direct program control command with one exception: the immediate data field actually points to another eight-word control block.

Figure 66. Cycle stealing input/output (part 2) (2 of 2)

- One intervention to start the process—by pointing to an immediate device control block containing the Start I/O command
- A sequence of device control blocks chained together with internal pointers. Each block contains a channel command and is read as needed by the device controller using the cycle steal channel.

In a sense, a sequence of device control blocks chained together is a program which controls a sequence of operations and transfers on the cycle steal channel. The channel controller is functioning much like a processor itself—and in fact it is a microprocessor.

The device control block is an eight-word information block whose contents are partially standard and partially device dependent as shown in Figure 67. Words one through five are dependent upon the particular device and the particular command within the control block. For example, a transfer to a disk would require cylinder and sector addresses as well as a main storage address and a byte count. The interface designer allocates the information needed by each device to these words in the control block. OEM cycle steal interfaces may be requested—to user specification—on a special order basis, that is a request price quotation (RPQ), from IBM. The GPIB Adapter card is a cycle stealing feature for OEM attachment.

The first word of the device control block is the control word whose fields are defined in Figure 67. Notice that the individual bits:

- Signal the controller commands, like interrupt, upon completion of data transfer
- Operate in burst (channel-dedicated) mode
- Chain to another device control block upon completion of this transfer
- Direct the transfer
- Perform error control

The seven-bit modifier is available as a device-dependent command field. The system uses the cycle steal address key

field during the transfer as part of storage protection and address translation as indicated in the section of this chapter entitled "Storage Protection and Address Translation Effects on Input/Output Operations". Notice again that this information field, along with others in the device control block, constitutes a complex command to the channel.

Burst mode means that the channel is dedicated to the data transfer during the entire time the transfer occurs. In burst mode, the processor and all other devices are completely locked out of the channel. If the transferring device is fast enough—the burst mode can accommodate very fast devices—data transfers occur at the maximum rate that the main storage can accommodate. Burst mode might be useful where the remote device is another computer (computer to computer communications) and the user wants control over concurrent accesses to a data base or critical data area. By locking the channel during a transfer, a computer can transfer data and assure the user that the processor will not simultaneously update the data. This procedure permits communications' software to resolve contention problems which arise in networking applications of small computers. The technique is analogous to the processor inhibiting or masking interrupts to prevent simultaneous access to a critical data area by other tasks within the processor itself.

The device control block resides in main storage itself. When the channel receives a Start I/O command, it interacts with the addressed device controller and fetches those words in the device control block needed by the interface to initiate and carry out the command. In this way, fetching the command itself uses storage accesses and cycles stolen from the processor. The system does not necessarily have to fetch all eight words contained in the device control block; it fetches only those words needed by the device. The exact number is then device dependent—in fact, it may depend upon the particular modifier field. For effective control over concurrency of task execution and input/output, the cycle steal operation, optionally, interrupts upon completion of each block transfer.

The operation of a cycle stealing device involves initiation by direct program control operations as indicated in Figure 68.

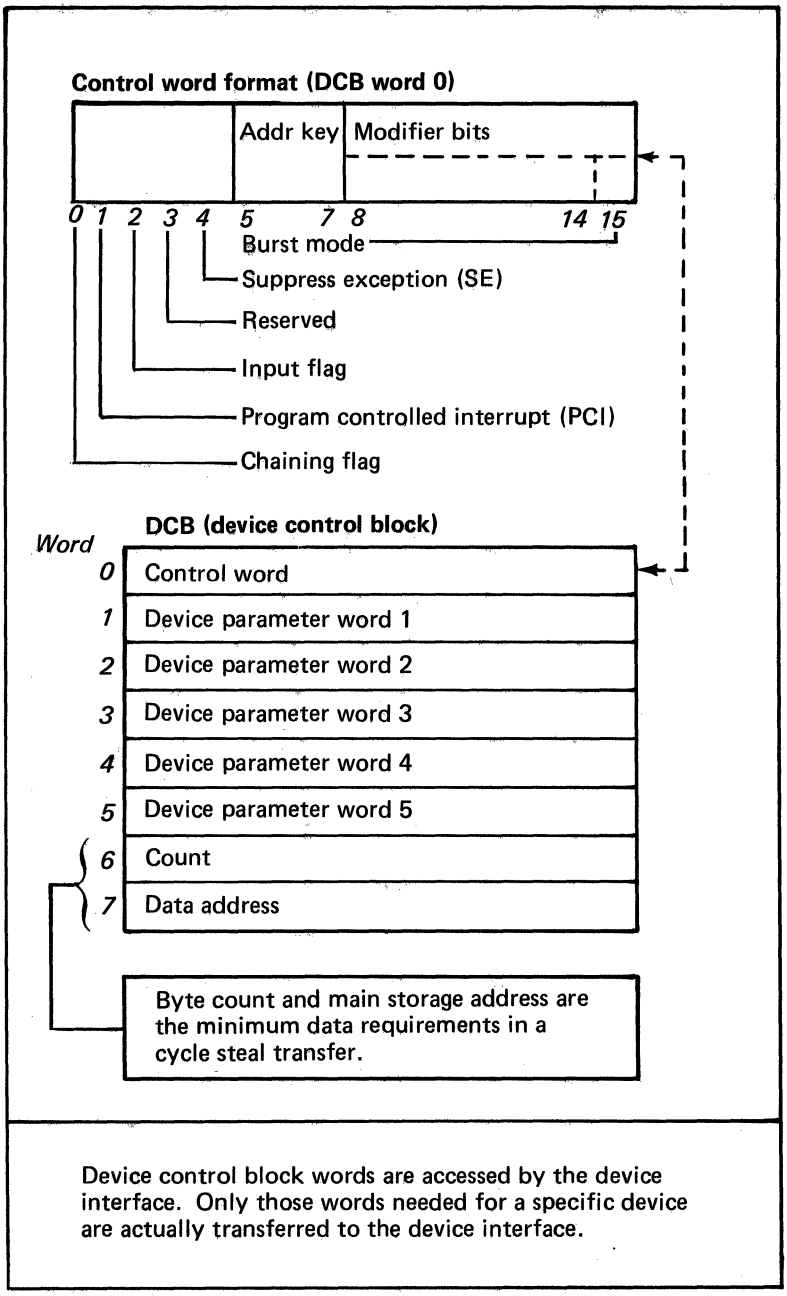


Figure 67. The device control block contains the data necessary to carry out one transfer between a specific device and main storage (1 of 2)

Bit 0	*Chaining flag.* If this bit is equal to one, a DCB chaining operation is indicated. After completing the current DCB operation, the device does not interrupt unless a program-controlled interrupt has been requested. Instead, the device fetches the next DCB in the chain.
Bit 1	*Programmed controlled interrupt (PCI).* If this bit is equal to one, the device presents a programmed controlled interrupt (PCI) at the completion of the DCB fetch. Data transfers associated with the DCB may commence even if the PCI is pending.
Bit 2	*Input flag.* The setting of this bit tells the device the direction of data transfer. 0 = Output (main storage to device or no transfer) 1 = Input (device to main storage) For bidirectional data transfers under one DCB operation, this bit must be set to one. For control operations involving no data transfer, this bit must be set to zero.
Bit 3	*Reserved.* This bit must be set to zero to avoid future code obsolescence.
Bit 4	*Suppress exception (SE).* If this bit is equal to one, reporting of device specified exception conditions are suppressed. The device continues the operation. The classes of exception conditions that can be suppressed are device dependent. For example: an incorrect-length-record condition could be suppressed. An exception that occurs during a DCB fetch operation cannot be suppressed. Refer to the individual device publications.
Bits 5–7	*Cycle steal address key.* This key is presented by the device during data transfers. It is used to ascertain storage access authorization.
Bits 8–14	*Modifier.* These bits may be used to describe functions unique to a particular device.
Bit 15	*Burst mode.* If this bit is equal to one, the transfer of data takes place in burst mode. This mode dedicates the channel to the device until the last data transfer associated with this DCB is completed. This bit is device dependent if burst mode is not supported by the device.

Figure 67. The device control block contains the data necessary to carry out one transfer between a specific device and main storage (2 of 2)

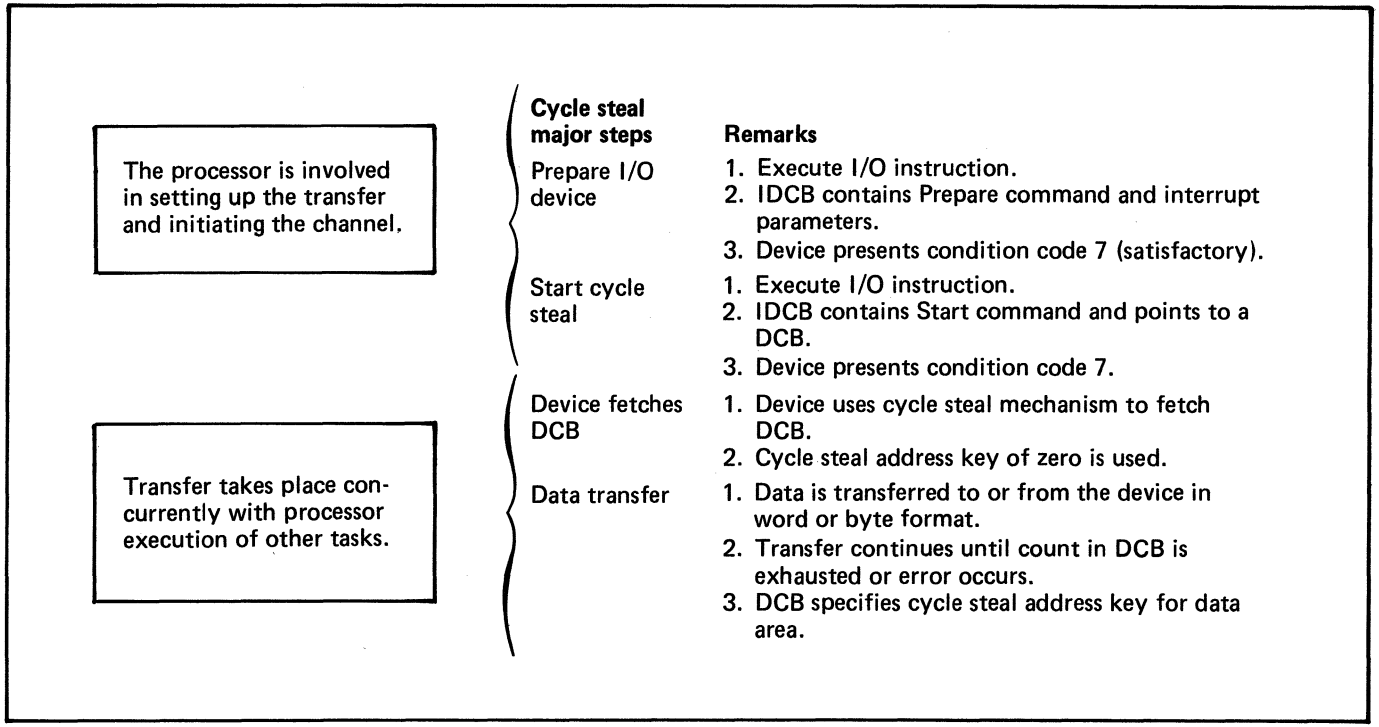


Figure 68. Sequence of operations during cycle stealing transfers (1 of 3)

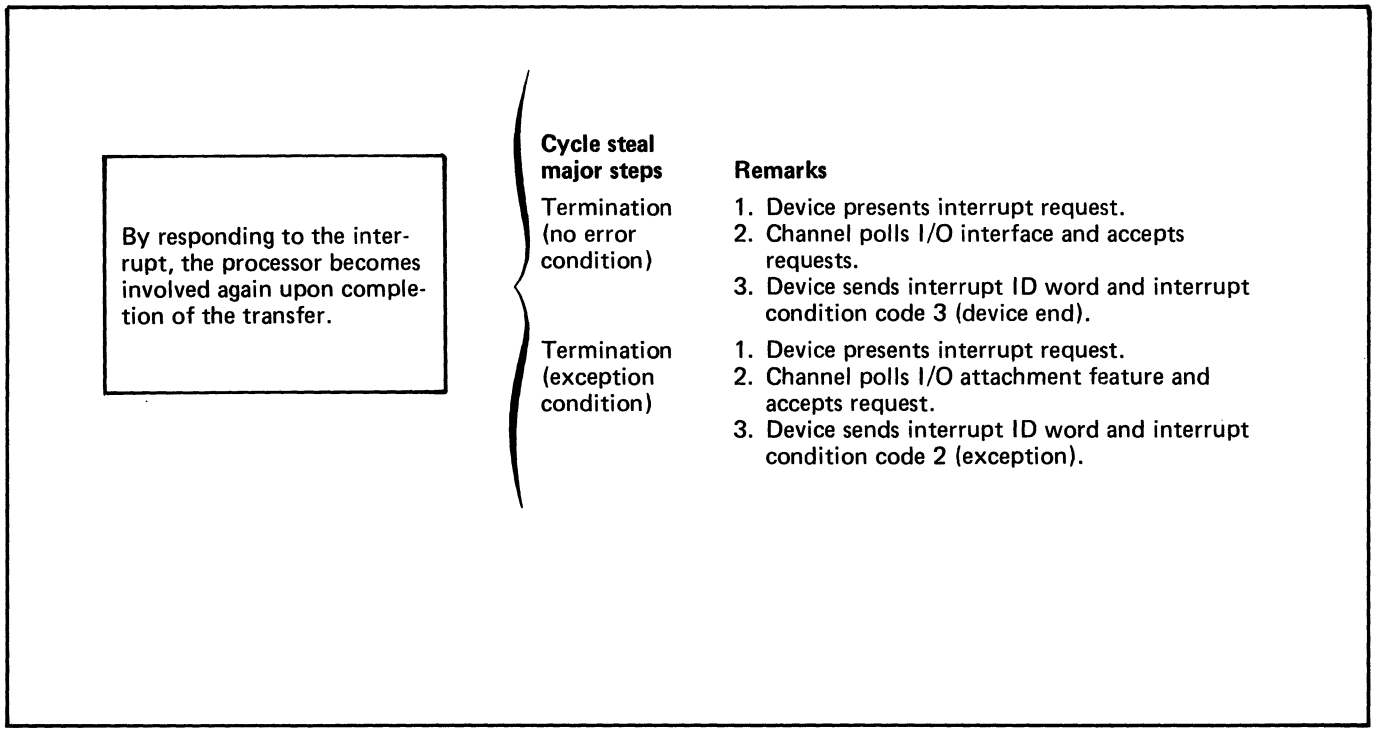


Figure 68. Sequence of operations during cycle stealing transfers (2 of 3)

Note: Other events that might occur during the cycle steal operation are:

Cycle steal major steps	Remarks
Chaining	1. Device completes the current DCB operation but does not present an interrupt request. 2. Device fetches next DCB in the chain.
Program controlled interrupt	1. Device fetches DCB (PCI bit = 1). 2. Device initiates an interrupt and sends an interrupt ID word and interrupt condition code 1 (PCI).

DCB: device control block (see Figure 66)

IDCB: immediate device control block (see Figure 66)

Interrupt I/O word: (see Figure 63)

PCI: program controlled interrupt

Figure 68. Sequence of operations during cycle stealing transfers (3 of 3)

Storage Protection and Address Translation Effects on Input/Output Operations

Storage protection on the Series/1 is performed in two basic ways: 1) in systems without address translation, by access keys associated with each 2K-block of storage translation; and 2) in systems with address translation, by access keys which address different stacks of segmentation registers. The input/output system is fully integrated into this system of storage protection (Figure 69).

Storage Protection Without Address Translation

For Series/1 systems without address translation (or those in which address translation is disabled), access is controlled as it is for any other instruction type which accesses storage: through the same address key register of the level on which the direct program control instruction is executed. Input and output operations use the operand 2 key to fetch the device control block. Cycle stealing, input/output operations are more complex because they involve fetching the device control block as well as reading or writing data in another area of main storage.

Reading and writing data in main storage during a cycle stealing operation requires a match between the address key of the block in storage and the address key stored in the control work of the device control block. This procedure is consistent with the conventional non-mapped storage protection procedure with one exception: the read/write protection bit is ignored during a cycle steal operation.

Both Series/1 hardware and software adopt a system-wide convention to handle the problem of accessing the device control block without violating storage protection. The key zero is used during access of the device control blocks (recall that a key of zero has no other special characteristics and that the no-protection key is key seven). With this convention, the hardware can set the address key register when an interrupt occurs because the system knows that key zero will be used to access the device blocks. In addition, input/output hardware and software can remain consistent with the storage protection objectives of the system.

Storage Protection With Address Translation

With address translation present and enabled, the accessing procedure is different. Tasks are protected from one another because their access keys determine sets of segmentation registers; these registers, in turn, map the task's 64K-byte address space into the physical main storage which is larger than 64K bytes. For operational efficiency and data protection, tasks performing input/output instructions need the same, consistent relationship as those accessing the device control block. For example, to ignore storage mapping and protection during an input/output operation would clearly limit how much error detection could be incorporated into the overall system—an unacceptable compromise in today's data processing world. If the available space is not contiguous, cycle stealing input/output operations into physical storage might require multiple commands to transfer data and program code. Such a procedure would be inefficient and unacceptable in a modern small computer.

A direct program control, input/output operation poses no additional address translation and protection problems. All the addresses are checked by the same mechanism that handles other-purpose storage requests from the processor. Because it is addressed by the I/O instruction and fetched, the immediate device control block must reside in the space mapped by the operand 2 key stack of segmentation registers. This structuring usually presents no problem because—except for special customized systems—all actual input/output instructions are carried out by the operating system, and all device control blocks reside within its space. This arrangement is consistent with the hardware convention of using address key zero both for fetching device control blocks and for initiating the operating system.

In cycle steal operations, the device control block must reside in the space controlled by the task executing the cycle steal command. By convention, the device control block is fetched with key zero (this block is the set of segmentation registers allocated to the operating system which also uses key zero); consequently, there is no problem in accessing the block. The data buffer may reside in any space

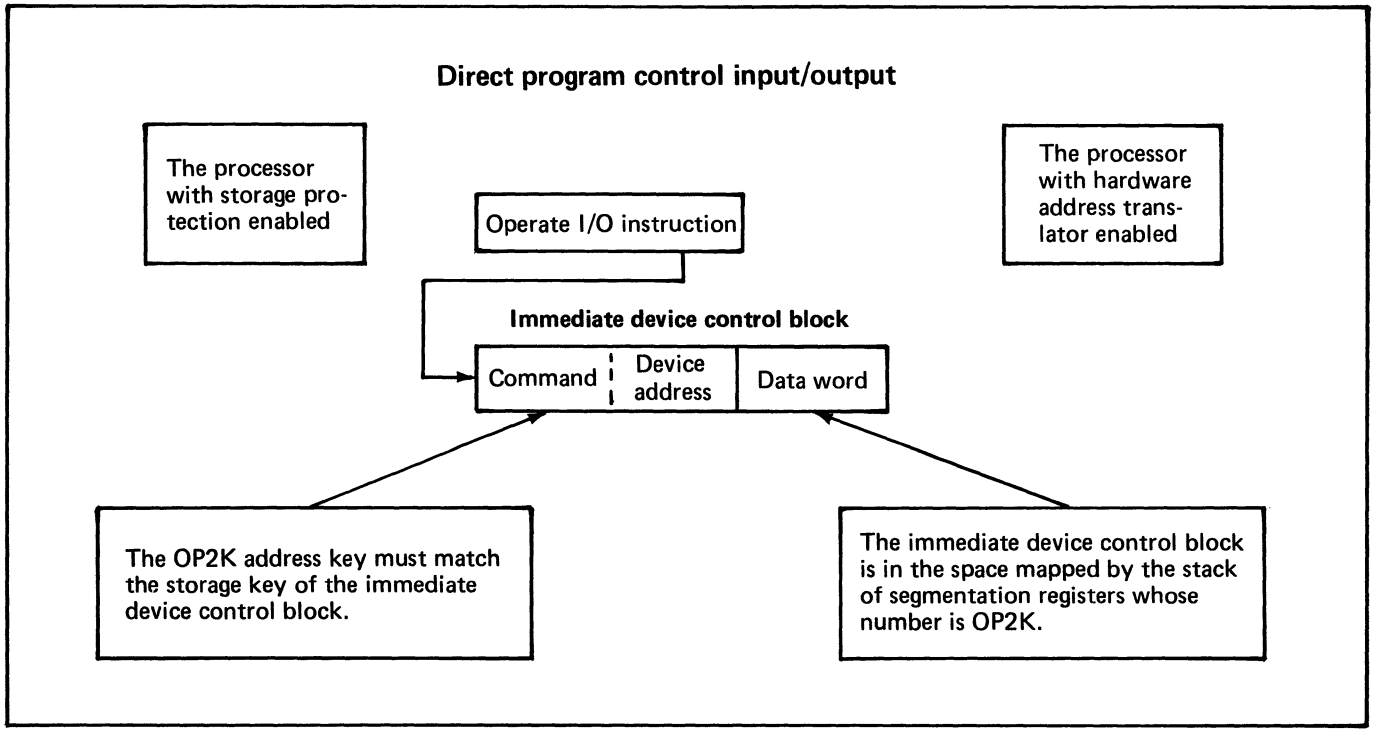


Figure 69. Input/output is consistent with storage protection of both mapped and unmapped processors (1 of 4)

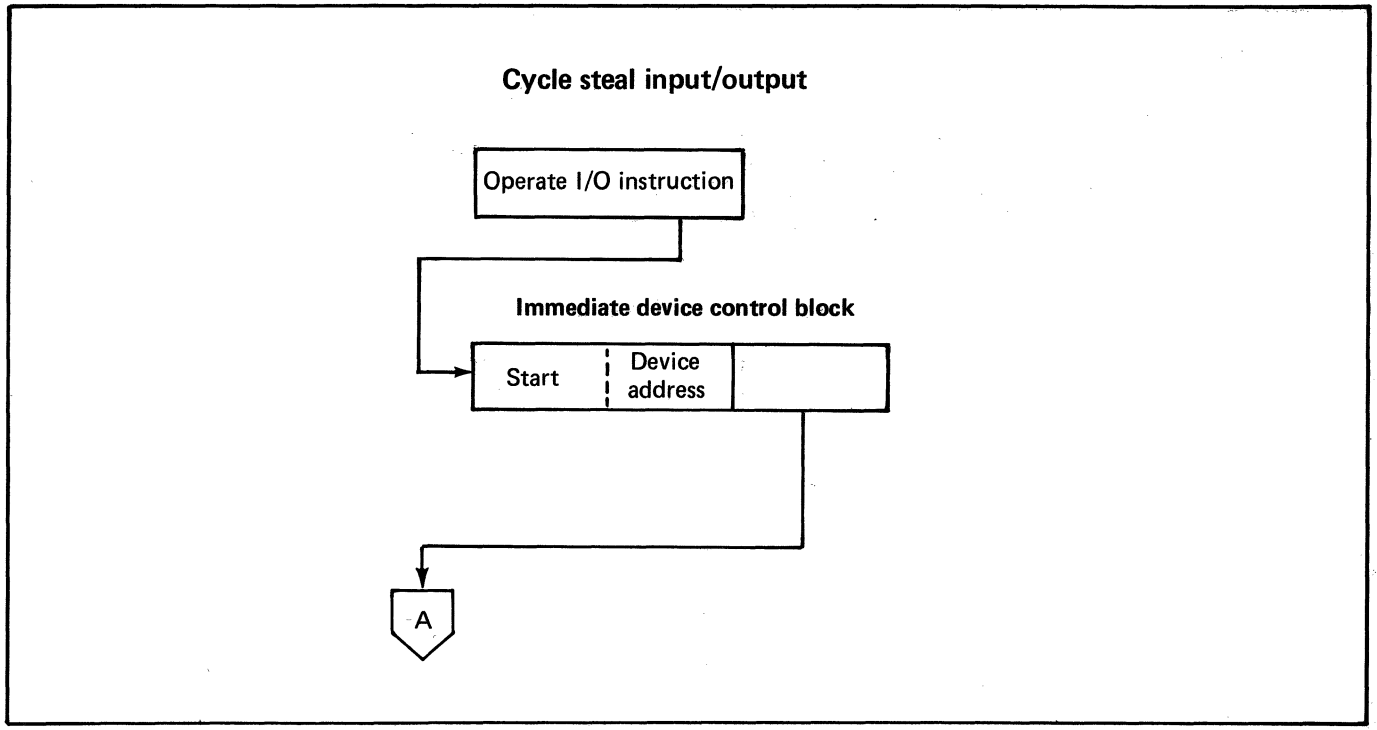


Figure 69. Input/output is consistent with storage protection of both mapped and unmapped processors (2 of 4)

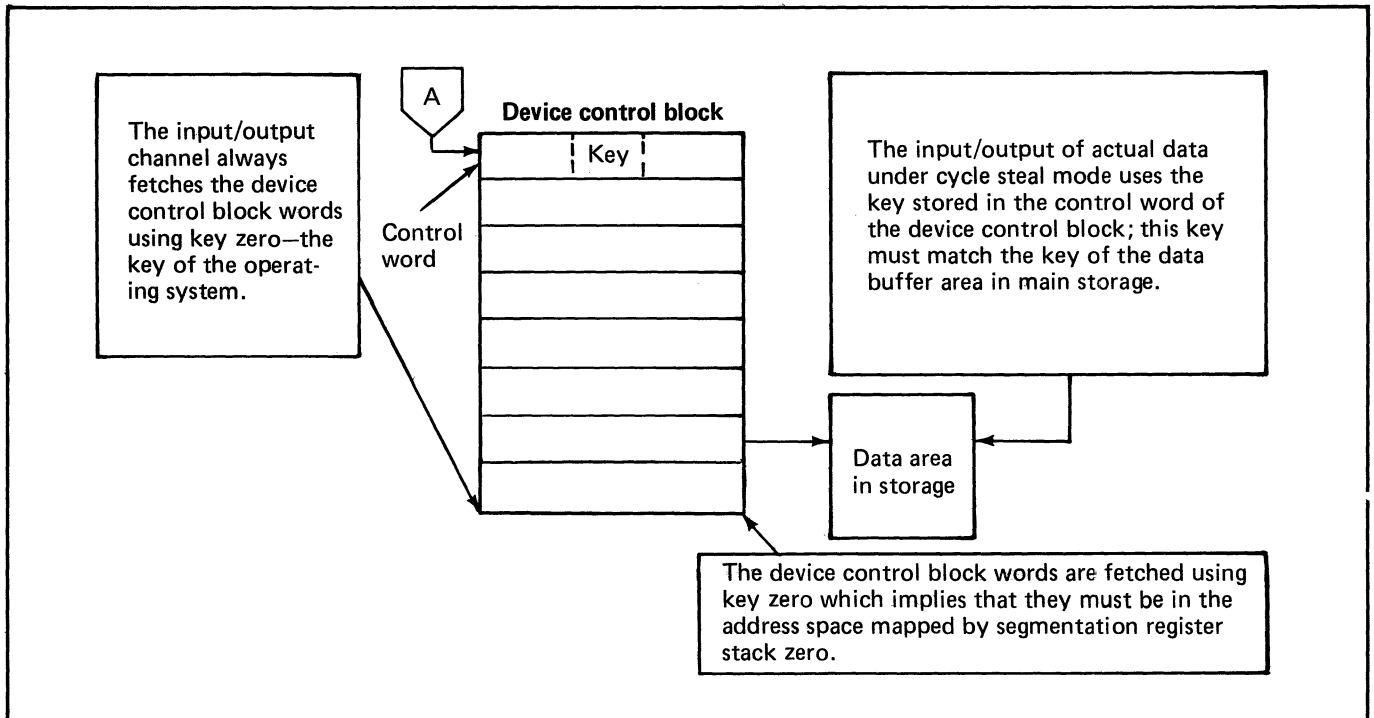


Figure 69. Input/output is consistent with storage protection of both mapped and unmapped processors (3 of 4)

Using the key and segmentation stack zero for accessing device control blocks is a hardware/software convention of the Series/1. This convention facilitates using the key and the address space for the operating system. To insure overall system integrity, the system performs input/output within the operating system.

Figure 69. Input/output is consistent with storage protection of both mapped and unmapped processors (4 of 4)

with any key. The particular key used is stored in the first word of the device control block. This organization permits a user task to initiate a cycle steal transfer into a user task data area by calling the operating system to do the actual initiation.

Whether or not the Series/1 is using address translation, the user can perform input/output operation in a straightforward, fully-protected manner with full use of system error checking. The only requirement is that the operating system or the supervisor program must maintain the device control blocks, safely, within its own area.

Software Use of Input/Output Hardware

The Series/1 architecture allows the system great flexibility when performing both direct program control and cycle steal input/output. Furthermore, storage protection and control over interrupts is maintained. To take advantage of this capability, however, an integrated system of software must be available. The Realtime Programming System, the Event Driven Executive, and the Control Program Support package provide user interfaces to:

- The input/output hardware integrated with error detection software
- Task management
- Other features typical of small computer software needs

Control Program Support of Input/Output

First, consider the Control Program Support routines for input/output operations. This package is a set of routines designed as a set of modules which can be used by application tasks to perform the basic functions of task management and input/output rather than being designed as a stand alone operating system. The user's ability to tailor software for a dedicated application is an especially important feature of the Control Program Support. The simplicity of software design makes it easy to understand the basic techniques used for task management and input/output in IBM-supplied software.

For purposes of system integrity, it is advantageous to perform all input/output operations in the supervisor mode. It is further desirable to isolate application tasks themselves from those privileged instructions being used to set up and carry out input/output. The special instruction called Supervisor Call (SVC) (Figure 70) performs all communications between tasks and: 1) the operating system (including its input/output functions), or 2) the Control Program Support package. This instruction includes a parameter field which specifies the exact service desired. Execution of this instruction causes an internal or class interrupt which is handled in the standard fashion described earlier: the level status block (registers whose contents must be later restored before returning to the user task) is stored in the system table. Each type of internal interrupt has a unique hardware location which must contain the address of the save area for the level status block. In the case of the service internal interrupt, this address happens to be location 16. As part of the same instruction, interrupts are inhibited and the system is changed to supervisor status.

To be consistent with storage protection, address keys are modified as shown in Figure 70: the source operand key OP1K is replaced by the value of destination operand key OP2K; then, OP2K and ISR are set to zero. The former permits the system to access the user's area to get information in buffers and perform similar operations. The latter is the Series/1 hardware/software convention which facilitates accessing of buffers and device control blocks in the system space (always key zero).

Finally, (and still part of the single-service instruction), the parameter associated with the SVC instruction is loaded into register one. Control then passes to the starting-instruction address of the service routine which is stored next to the address of the level status block save area in location 18.

Notice that this single instruction does the following:

1. Performs all of the system's housekeeping tasks like the saving of registers

2. Sets the system up for the branch to the appropriate service routine by loading the parameters of the SVC into register one

The user task takes advantage of this instruction to link to a service module as depicted in Figure 71. The Control Program Support package assumes that all information transferred between the user task and the service module(s) is done through a block of information called the IOCB: input/output control block. To perform an input/output operation for a specific device, the address of the control block for that device is loaded into register zero. Then, the SVC instruction is executed with the parameter set to the desired service. Following the hardware functions described above and in Figure 70, the system transfers control to the service routine module which sets the device status to busy. If the device is already busy, the request is queued just as it is in the Realtime Programming System. The service routine then branches to the appropriate module and carries out the function indicated using the parameters and the data contained in the addressed input/output control block.

For users to maintain full control over the application, they must be able to specify whether or not input/output is overlapped with further computation of the active task or with some other task. To this end, control is immediately returned to the user task as soon as the requested input/output operation has been initiated (the system automatically restores the level status block and returns to the non-supervisor mode). Register zero is loaded with a code indicating success of the SVC, or failure—an error occurrance. If successful, the user then has two options:

1. To continue execution, overlapped with the input/output operation
2. To use the tasking facilities to suspend task execution until the system completes the input/output operation

As part of the input/output control block information, the user specifies whether or not a special user task is to be executed upon completion of the operation. As shown in

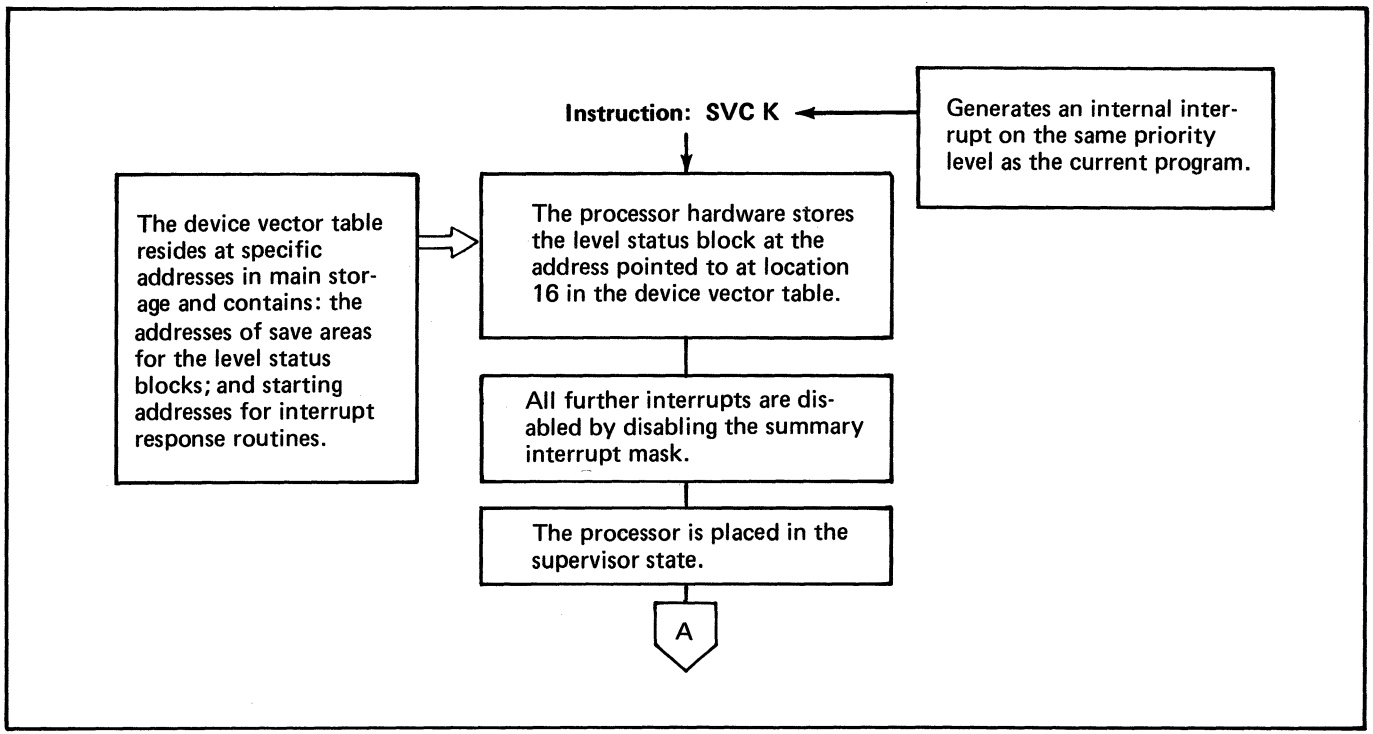


Figure 70. Communications between a task and the operating system using the Supervisor Call (SVC) convention (1 of 3)

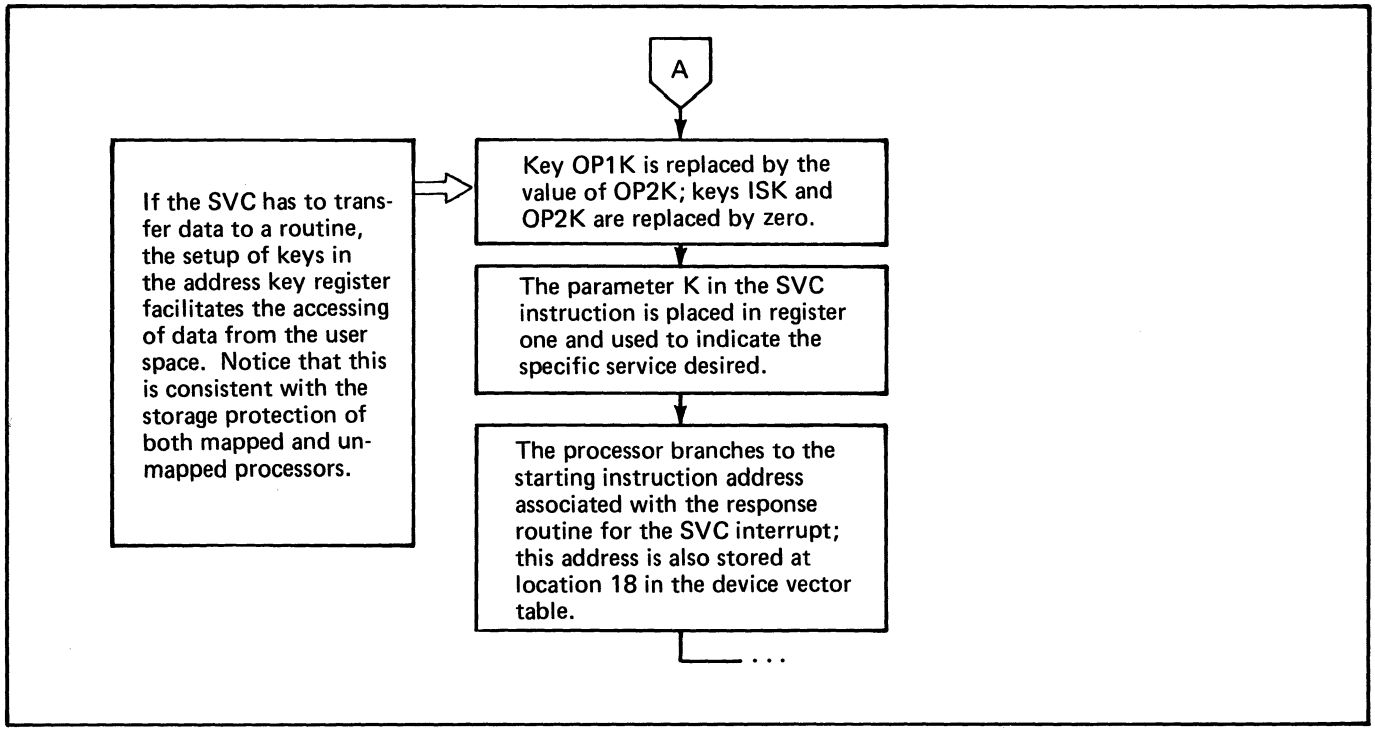


Figure 70. Communications between a task and the operating system using the Supervisor Call (SVC) convention (2 of 3)

Communications between a task and the operating system are through a special instruction—SVC—which generates an interrupt.

The entire linkage—from the generation of the SVC interrupt to the branching to the first instruction of the interrupt response routine—is done by *hardware* alone and takes approximately 14.3 microseconds—this includes the time required for the storage of the registers in the level status block.

Figure 70. Communications between a task and the operating system using the Supervisor Call (SVC) convention (3 of 3)

Figure 71, upon completion of the input/output operation, the user can suspend execution and initiate a special task which can then continue the suspended task. Alternatively, the user need not supply such a special task. In this case, the system indicates completion of the input/output operation in the control block but takes no further action. If users must know whether or not the operation is complete, it is their responsibility to test the indicator in the control block.

Notice that the initiation of a special task is analogous to the generation of a user-created internal interrupt and gives the user complete control over the execution of tasks and input/output operations.

Input/output functions or services available are shown in Figure 72. The connect function sets up a device for subsequent input/output operations. Prior to calling for connect, the device is attached to a special null interrupt handler routine. Disconnect is the inverse function and is carried out upon completion of input/output operations—which in turn frees the device for use by another task. The read and write services input and output data into buffers whose addresses are defined in the input/output control block. All of these IBM-supplied routines perform extensive error checking, including multiple retries when they are appropriate. Depending on the source of the error, the system maintains an error log on a disk or diskette which includes information like:

- Error code
- The program status word
- The level status block at the time of the error
- Interrupt level and status byte
- Device address
- Condition codes

Error recovery is always possible through optional exits to user-supplied routines.

The Control Program Support package is intended to permit the user to tailor efficiently special-purpose, dedicated

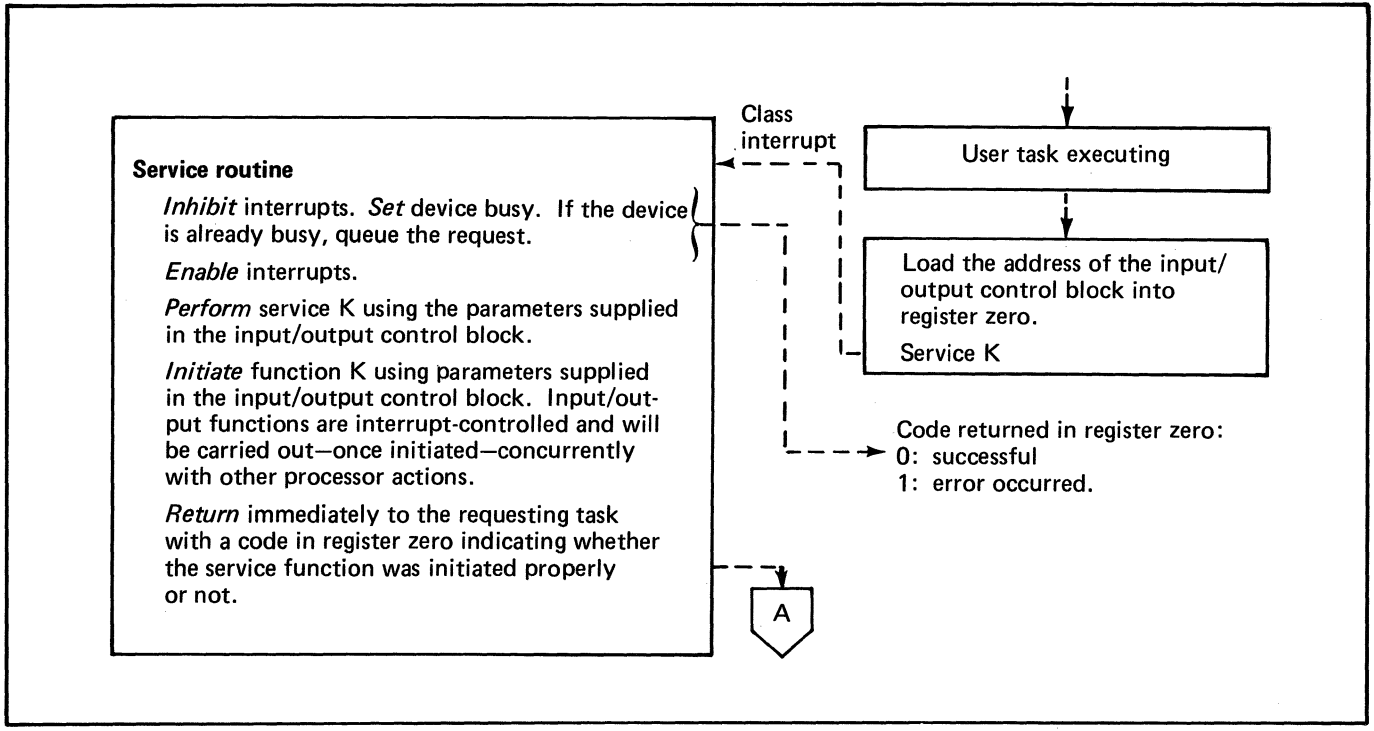


Figure 71. Overlapping and non-overlapping of input/output control (1 of 2)

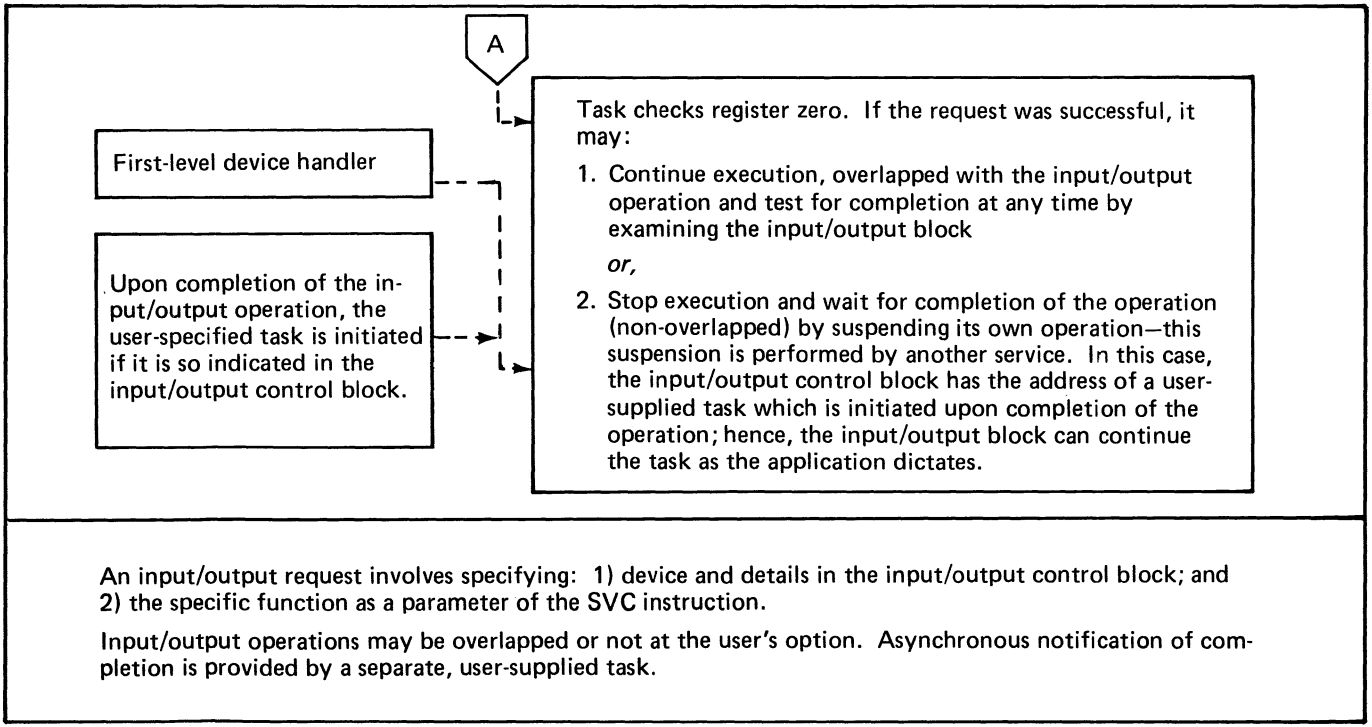


Figure 71. Overlapping and non-overlapping of input/output control (2 of 2)

- Connect—initializes the device vector table (the address of the device descriptor block and the starting address of the interrupt response routine). Prior to connect, all interrupts are connected to a dummy response routine.

  The device descriptor block is linked to the input/output control block in which all parameters and device-specific information is stored. Connect returns a code to the user indicating whether or not the request was carried out.
- Disconnect—breaks the connection between a device and an input/output control block.
- Read—a service called to input data from a device or file. Parameters in the input/output control block are validated, and control is passed to the appropriate device routines to initiate the transfer into the caller's buffers.
- Write—a service called to output data from a user's buffer to a device or file. Parameters in the input/output control block are validated and control is passed to the appropriate device routines to initiate the transfer.
- Error log—a service to connect a file for error logging purposes. Errors are extensively checked and recovery tried in all Control Program Support routines. Errors are logged to the file whenever: a null interrupt is received; a recoverable error occurs during an input/output operation; or a nonrecoverable error occurs. Utilities for displaying the error log are available.

Figure 72. Input/output functions available in the Control Program Support package (1 of 2)

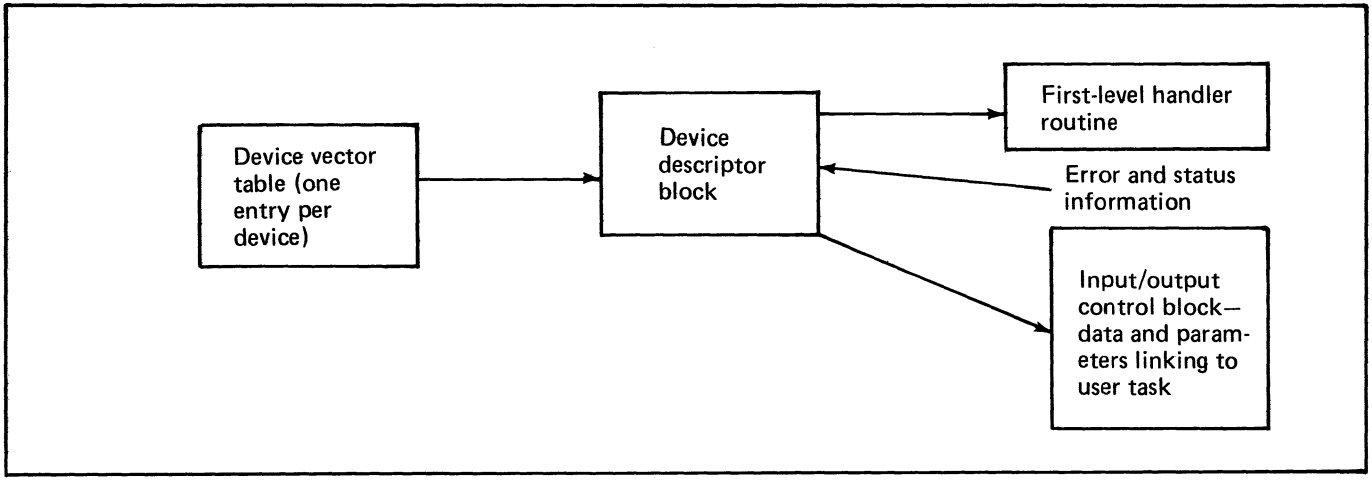


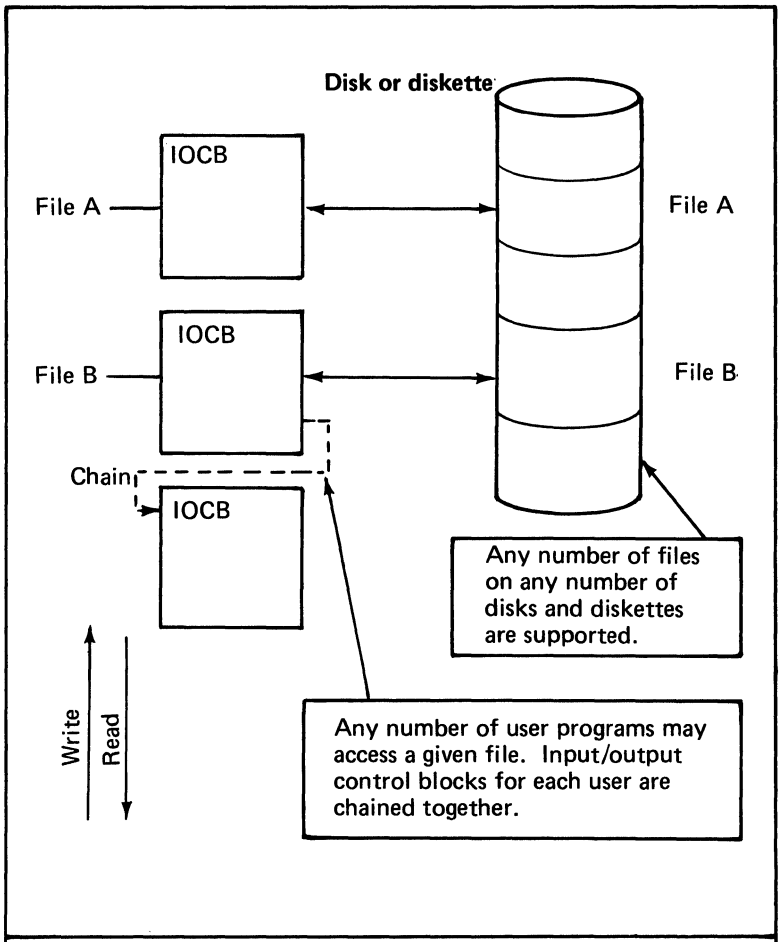
Figure 72. Input/output functions available in the Control Program Support package (2 of 2)

applications. This means that the user can conveniently link the special error recovery or special event-handling routines into the system without modifying the basic code of the package. An example of a common, critical small computer application need is the efficient handling of disk files. An elaborate files capability is available through the Realtime Programming System; but sometimes it is desirable for a user to lay out special-purpose file areas on disk. These areas should be accessible without passing through directories—with their consequent overhead.

In certain dedicated small computer applications, the security introduced by central file handling is less important than speed of access. Figure 73 shows how the Control Program Support package does this accessing. The user allocates areas to disks—specifying starting and ending, sector address, input/output control blocks—one for each special-purpose file area. The same functions used for other input/output operations are then available for reading and writing in those file areas. Because access is by relative sector, the user can control any blocking, logical record-length problems, and similar considerations in a completely dedicated, application-dependent manner. Most importantly, the system uses standard-access software so that none of the following functions are sacrificed:

- Error detection
- Error recovery
- Error logging
- Self-diagnosis
- Other Series/1 integrity features

The use of the Control Program Support package follows the general schematic shown in Figure 74. The user prepares a system's area module containing all the tables, device sector tables, definition of services, and other features depicted in Figure 74. The routines making up the control package are then present and linked through the tables in the system area module. Finally, the system prepares the user application tasks—including any special tasks which are initiated when events are detected by the control package



A file is allocated space on the disk by the user, not by the system. The file is defined by device address, starting sector, and ending sector number.

Access to the file is through Read or Write input/output service calls. The starting sector (relative to the beginning of the file) is specified along with the number of bytes to be transferred and the address of the user's buffer area. Access by relative record is random.

Input/output control blocks contain a relative sector field. Each access updates this field; a user can reference this field to use the file in a sequential mode.

Figure 73. Access to files using Control Program Support

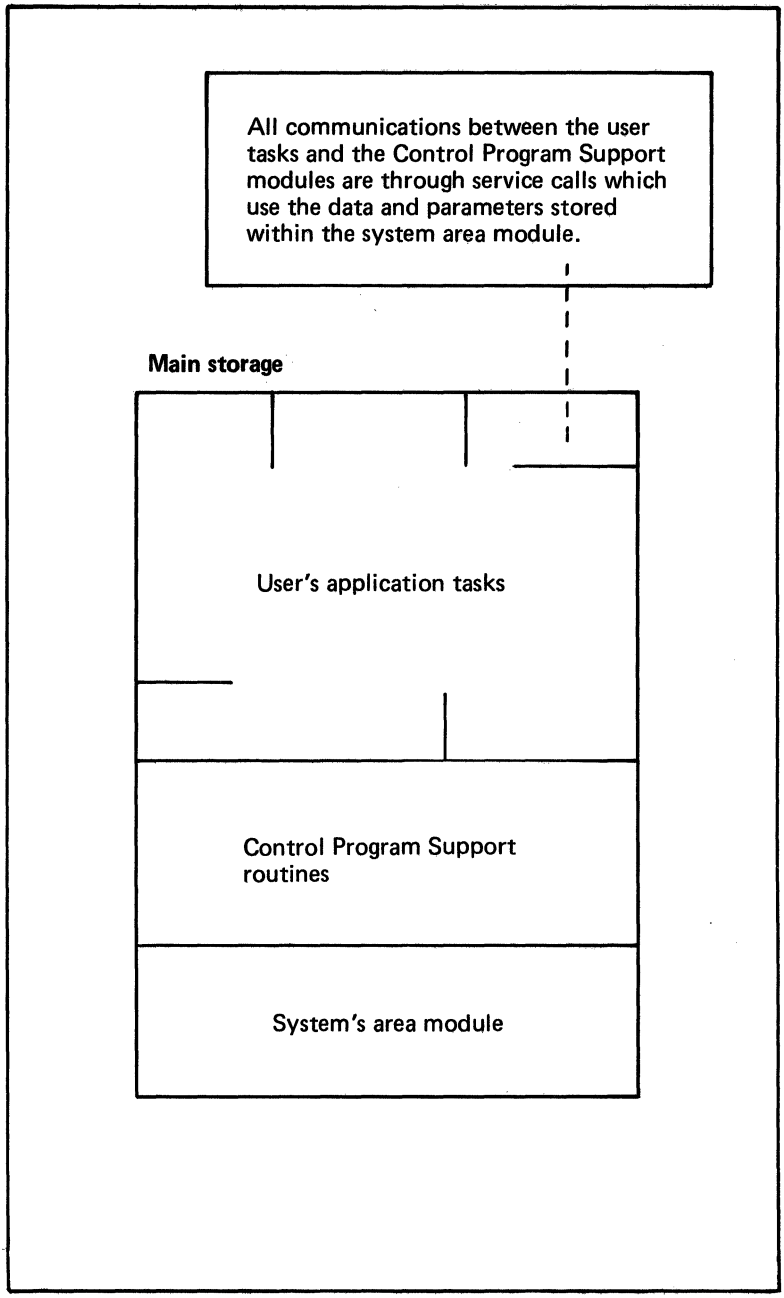


Figure 74. Organization of main storage for a dedicated application utilizing the Control Program Support package (1 of 2)

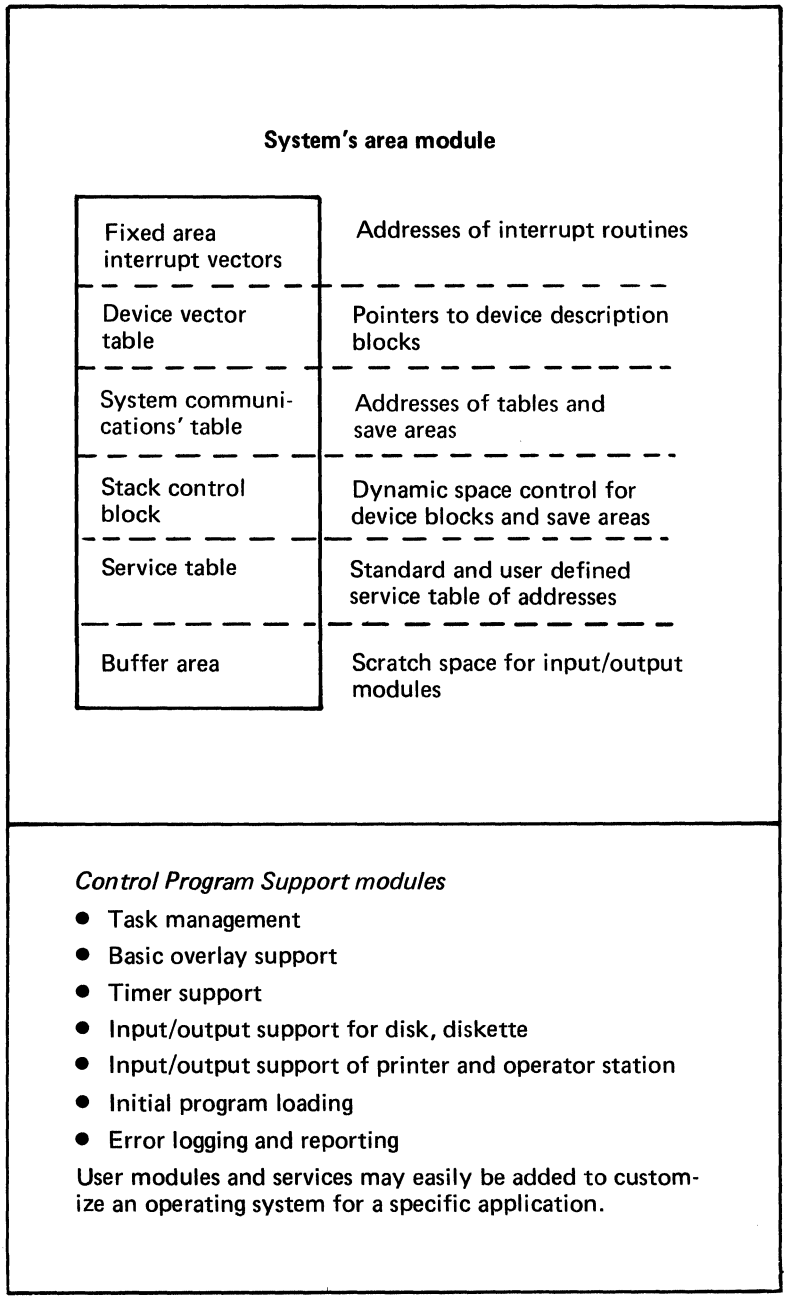


Figure 74. Organization of main storage for a dedicated application utilizing the Control Program Support package (2 of 2)

modules. The net result is a dedicated task in which the combination of Control Program Support modules and the user-supplied modules constitutes a tailored operating system.

Operating System Support of Input/Output

The Realtime Programming System makes available a higher level of software support for input/output operations. At the lowest level within the programming system is basic access which is essentially equivalent to that of the Control Program Support package input/output. This software level allows the user as complete a control over input/output devices as does the hardware. Furthermore, the system provides a set of macros, (pre-coded routines), to simplify the specification of device descriptor blocks, input/output control blocks, and other system tables needed to specify input/output at this basic level. These macros facilitate special, user-written software for OEM devices.

The user can also deal with input/output operations at the logical file or data set levels of the Realtime Programming System input/output software support. A data set or file is a collection of records of fixed or variable length, possibly grouped into blocks to expedite physical, input/output operations. Figure 75 illustrates four data-set organizations supported under the Realtime Programming System. The consecutive organization is simply a set of logical records grouped into blocks so the system can access the records only in a sequential manner. Physical devices, like line printers, can be treated as a consecutive data set if the user considers each line to be a logical record. Outputting data to a line printer becomes logically equivalent to outputting the same information to a consecutive file.

The random data set organization allows direct access to records by name or key using a technique which translates the key—for efficiency in locating the record—into an address on a direct access device. The index access method also provides keyed access to user data. The partitioned data set is simply a group of data sets with a directory. A data set within the partitioned set may be accessed by name through the directory and, once located, by one of the available

access methods (sequential or direct). Partitioned data sets are useful for libraries of routines, data sets, programs, and similar items accessed by name. Random data sets are very important in online small computer applications where: 1) access to a record must be rapid, and 2) the user cannot control the sequence of accesses.

When device input/output operations are treated at the data set level, the user can write software using logical rather than physical device names. Users can assign a physical device to a logical device name at execution time which provides the system with an economic way to adapt to changes in load, configuration, and device failures. Furthermore, the user can more easily debug applications by using data sets as sources and destinations of input/output operations (where the values can be controlled and checked for correctness).

Access to logical data sets (including devices) is at two levels: Read/Write and Get/Put. The Read/Write level inputs or outputs a physical block of data. The block may contain logical records or may be a single record; in either case, it is the user's responsibility to handle the block once it is obtained. In contrast, the highest level of access is Get/Put where reference is to logical records.

At this higher level, the following questions are handled automatically by the input/output routines within the operating system and are transparent to the user:

- All problems of breaking blocks into logical records
- Saving and restoring partially filled blocks
- Similar functional problems

The system provides full queuing of requests for input/output at the logical level together with user specification of overlap with task execution.

Generally, the higher the level of input/output support is:

1. The farther the user task is from the physical hardware of the input/output system, and
2. The source data management is done by the system rather than—explicitly—by the user task

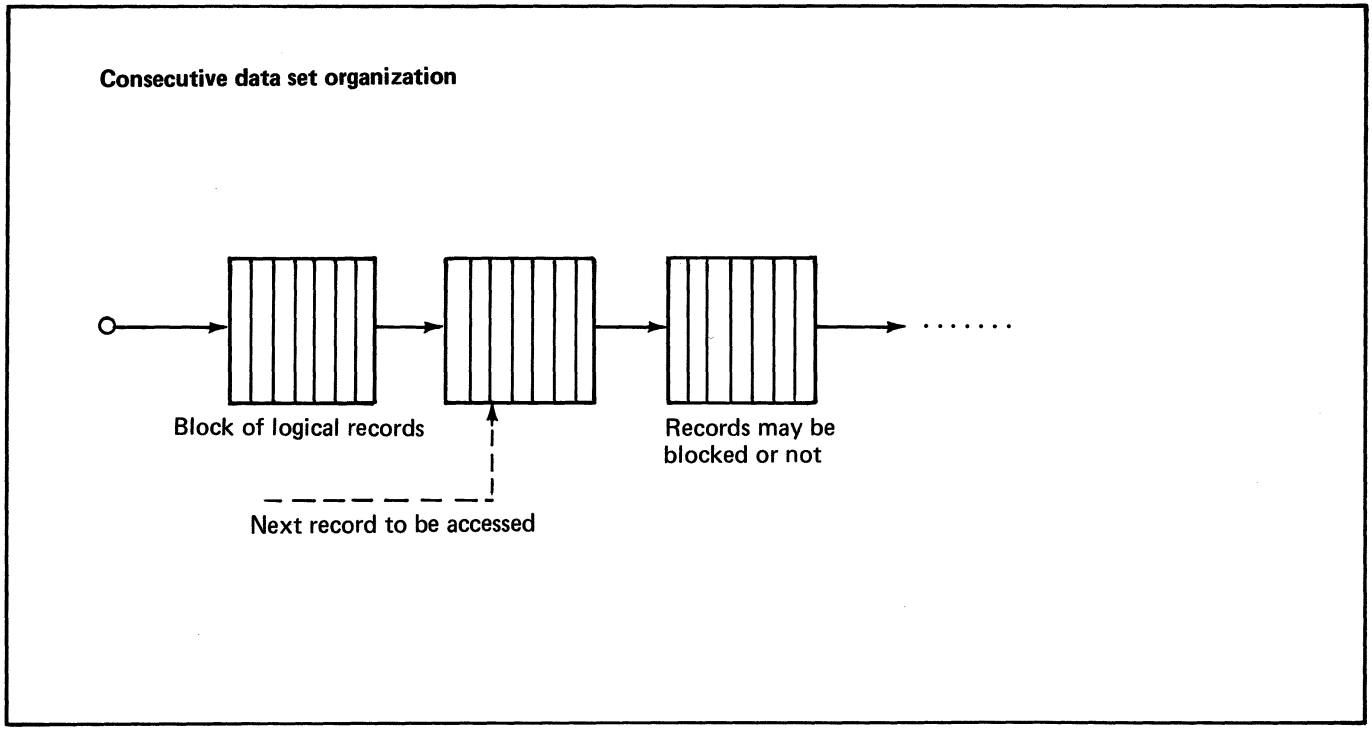


Figure 75. Four data set organizations supported under the Realtime Programming System (1 of 5)

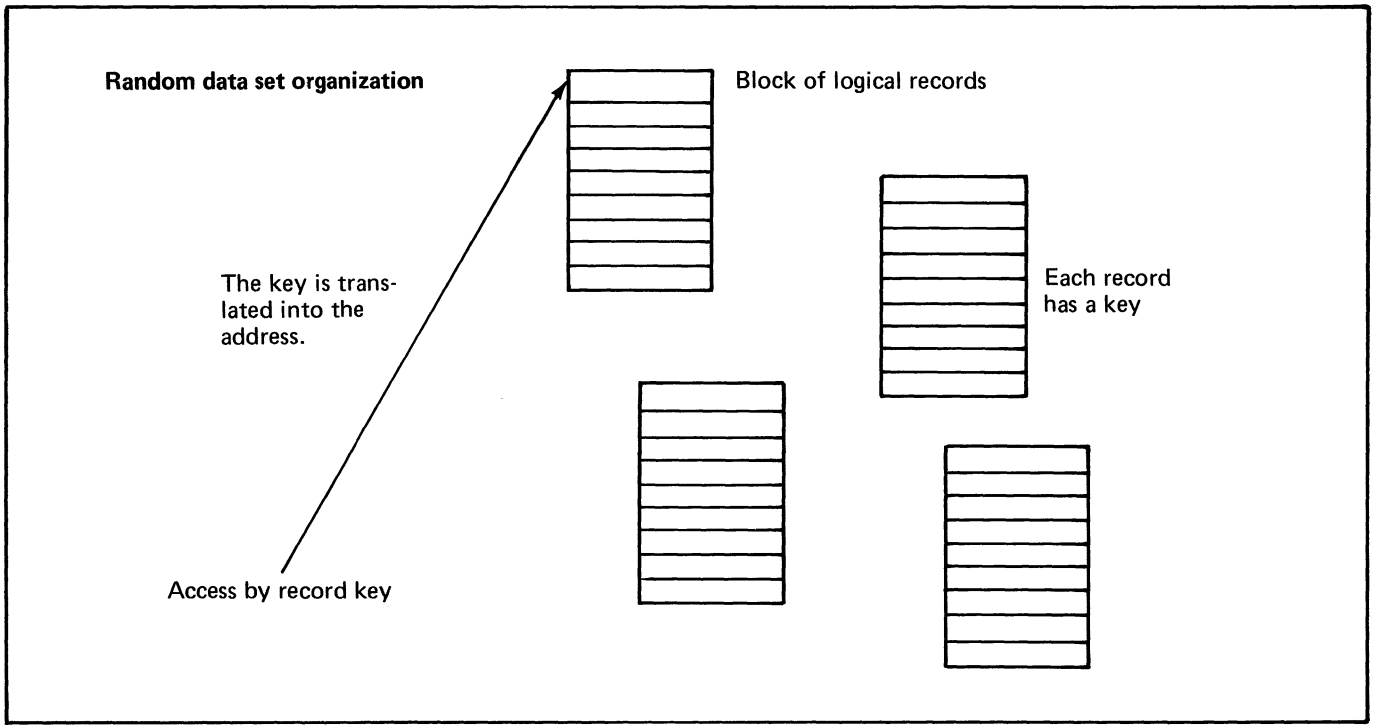


Figure 75. Four data set organizations supported under the Realtime Programming System (2 of 5)

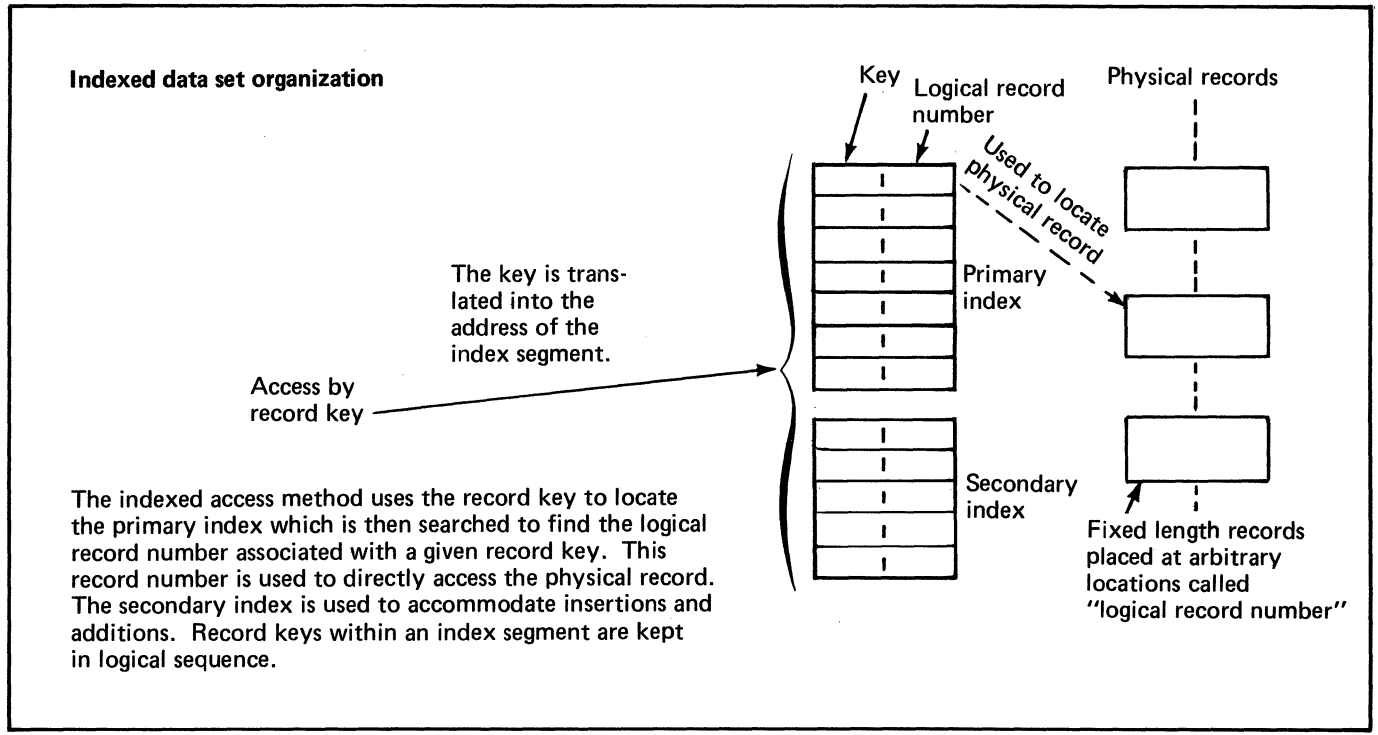


Figure 75. Four data set organizations supported under the Realtime Programming System (3 of 5)

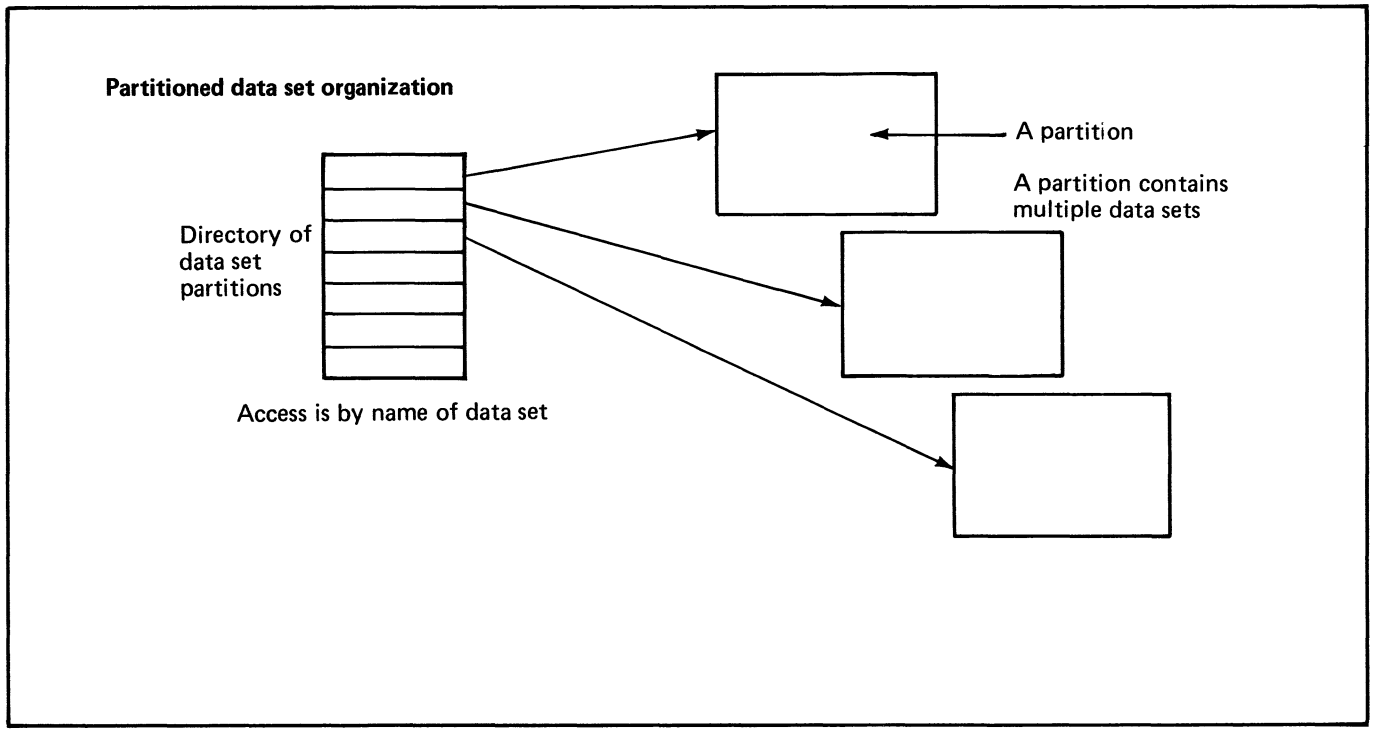


Figure 75. Four data set organizations supported under the Realtime Programming System (4 of 5)

The Realtime Programming System provides a very general file or data set support with access either:

- Direct
- Sequential, or
- By name

Data sets may be defined in assembler language programs using a complete set of macros to simplify the programming.

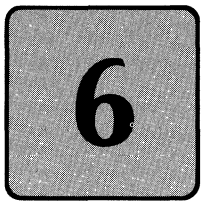
Access to data sets may be by block with Read/Write statements or to the logical record through Get/Put calls.

Data sets may be accessed either from assembly language programs or higher level language programs written in PL/I, COBOL, or FORTRAN.

Figure 75. Four data set organizations supported under the Realtime Programming System (5 of 5)

Each input/output level has its distinct place in the applications' structures and is necessary in an integrated hardware/software system. It is important to note that access at these levels is most compatible with the format of higher-level programming languages like PL/I, COBOL, and FORTRAN, all of which support the Read/Write and Get/Put levels of data set accessing and input/output device management.

In preparing an application task set, the user defines data sets, their characteristics, and the devices to be used. Access to these items is then specified according to the application need, and the access is programmed in the appropriate level language. As a result of these procedures, users gain a wide area of freedom in which to implement their applications, taking full advantage of the hardware of the input/output system and simultaneously utilizing several levels of standard software.



The Instruction Set and Its Use

Although effective realization of small computer applications is most sensitive to the organization, structure, and management of the processor, main storage, interrupt system, and input/output channel, it is also affected in a less obvious way by the design of the instruction set of the processor. This influence occurs, in part, in a negative way because consideration of system and application software needs has not been properly considered during the hardware design of many small computer systems.

The Series/1 instruction set was selected to efficiently support the integrated hardware/software system organization designed to service small computer applications.

Several areas of the overall system are especially critical and need the support of a strong instruction set for efficient implementation in the small computer application environment.

High-level Languages. The instruction set must permit efficient translation of high-level languages to enable those applications—whose programming is most appropriately done in FORTRAN, PL/I, COBOL or similar languages—to produce tasks equally efficient in both storage utilization and execution time. Furthermore, programs written in such

languages should be able to take efficient advantage of modern, structured programming techniques.

Realtime Programming System. The instruction set must allow the user to realize an efficient operating system that both preserves and enforces overall system integrity and protects shared tasks and data areas in a user environment of multiple, cooperating tasks. The instruction set must support fast task switching, reentrant programs, error detection, and other critical areas which the designer has built into the system hardware.

Critical Assembly Language Level Operations. Some applications are critical either in their throughput needs or in the detailed nature of their computations. Many applications need similar capabilities in small but critical sections of the task set. The instruction set must support these needs, including:

- Individual bit manipulation
- Logical operation
- Masking
- Special arithmetic operations like unsigned or multiple-precision arithmetic

Because these needs are difficult to meet in a small instruction set, processors with such sets are generally limited in the applications and environments they can support. However, the Series/1 is a microprogrammed processor which allows the implementation of a rather large and complete instruction set containing more than 160 instructions.

Microprogramming is especially powerful when the system uses complex instructions. This is so because microprogramming permits the user to include those instructions which are especially valuable in small computer applications without attendant increases in hardware costs. For example, manipulation of strings of characters is common in small computer applications like word processing, direct input of data, or CRT display. Large processors like the IBM System/370 provide instructions which manipulate these

strings efficiently (move the entire string, compare two strings, or scan for a specific character). IBM has included similar instructions in the Series/1 processor because they are important if application tasks are to be compact and efficient. However, tasks using these complex instructions often have a lower priority than other tasks, and their execution must not adversely affect the overall system response to external events. Using microprogramming, these instructions are made interruptible on the Series/1 so as not to delay the higher priority tasks. This is just one example of the integrated hardware/software design of the Series/1 processor.

The purpose of this chapter is to discuss the instruction set of the Series/1 and its use in system software and application software. The appropriate processor manuals describe individual instructions in detail; they are not fully defined in this book. The user must consult those manuals to determine details like:

- The exact format of an instruction
- The instruction's effect on registers and indicators
- The instruction's execution time
- The effects—on the instruction—of errors detected

Users should be familiar with the overall structure of the instruction set so that they can understand and efficiently take advantage of system software and higher-level languages.

Figure 76 shows five areas into which the instruction set can be classified. Data movement, arithmetic and logic operations, and status testing instructions form the bulk of most application tasks. These instructions are important for efficient translation of higher-level languages and for efficient realization of critical computational tasks.

Instructions associated with structured programs and control over concurrency become important when:

- Applications are structured into a set of cooperating tasks
- The tasks are themselves a structured set of modules which are, perhaps, shared among tasks

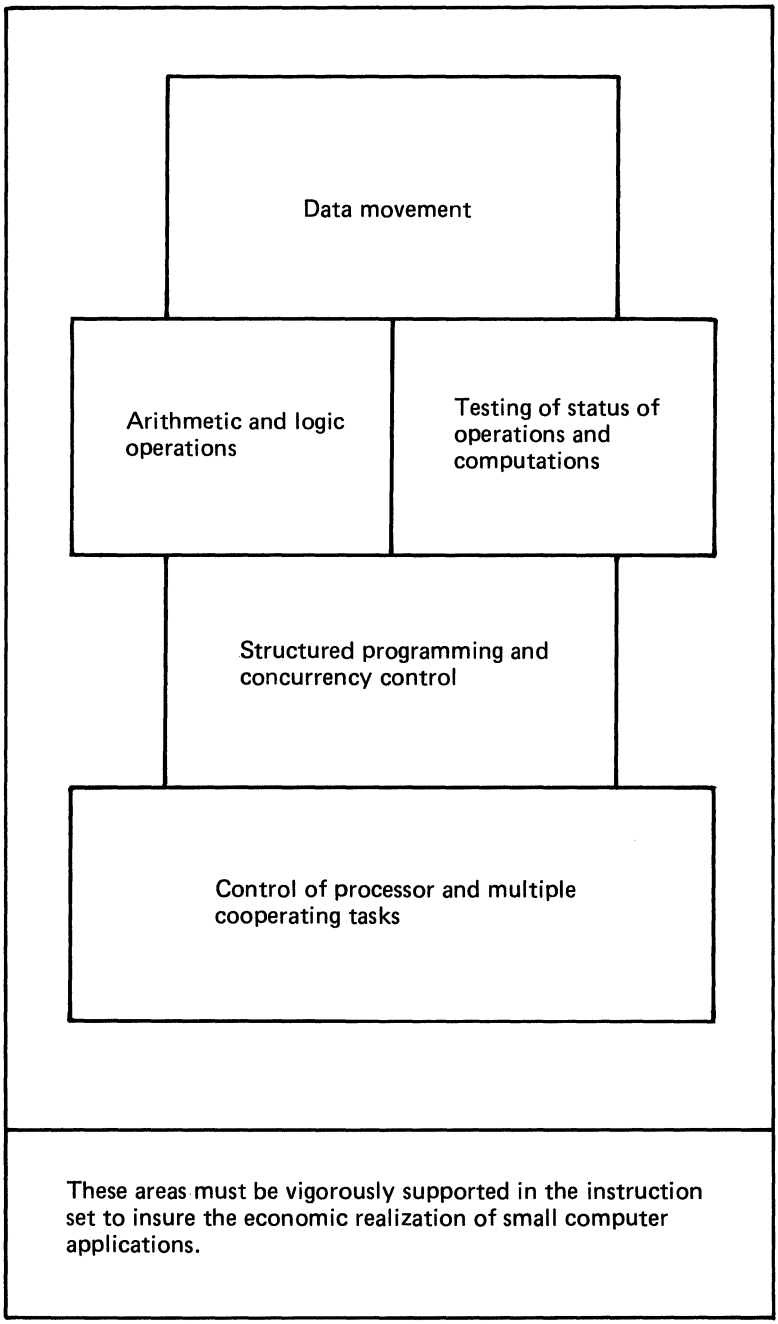


Figure 76. The five areas into which the instruction set can be classified

The instructions in this set of cooperating tasks support control over shared data and routines, simultaneous access, and other entities.

Finally, the instruction set must exercise control over system resources and the set of cooperating tasks competing for use of these resources. The last set of instructions is the one which controls processor status, registers, storage allocation, and other critical resources. It is this set which permits control over system integrity, response to errors detected in a task, and control over concurrent realtime events. The operating system is the entity that most often uses these instructions. However, critical dedicated applications may also use them. Clearly, this set of instructions primarily affects the efficiency of input/output routines, task switching, interrupt response, and servicing of application task requests.

This chapter covers, briefly:

- Each area of the instruction set
- The overall format of instructions
- The implementation of the various addressing modes previously introduced

Instruction Formats

An instruction is a command encoded in one or more words in storage. Depending upon the particular command, additional items of information must be supplied along with the command. These items include identification of registers to be used, data addresses, and/or immediate fields, and others. The problems involved in determining which specific instructions to include in a machine occur in two areas: 1) selection of the commands to be included, and 2) packing of the information needed for each command into the limited space available in a word in storage.

The format of an instruction is the detailed specification of which information is packed into which bit field of the instruction. System designers might select a particular instruction to enhance simplicity of understanding, readability of object code, or other esthetic considerations.

These considerations were *not* the objectives of the Series/1 design. The Series/1 instruction set has a variable-length design because of:

1. The variety of instruction commands desired
2. The variation in number and length of additional information items required to specify function, operands, and addresses

This design results in one- two- and three-word instructions.

Some commands require no operands at all. Examples are: instructions which halt the processor (Stop), enable interrupts, interchange operand keys, and similar operations. In these examples, one word is more than adequate to encode the commands. These and other instructions are termed *parametric* because, while they do not refer to data operands in storage or registers, they often include a field which particularizes the command or selects one specific command from a group of commands.

Commands may include data—called immediate data—within the command itself. If this data is short enough (a byte for example), it might also be stored within one instruction word. If the immediate data item is itself a word in length, the instruction must occupy a minimum of two words.

Commands which reference operands in storage may do so in a variety of ways called *addressing modes.* If the data item itself or its address is in a register, the instruction need contain only the register number (a two- or three-bit field); this field may be contained within a one-word instruction. However, if the operand must be addressed using base relative addressing, the displacement from the base register may be too long to fit in the instruction word; in this case, the displacement would be stored in a separate, second word of the instruction.

Instructions which reference two operands in main storage may then require three-word instructions to contain all the addressing information. It is the responsibility of the programmer and the program-support software to attempt to use addressing modes and program organization which minimize instruction lengths. The Series/1 assembler, for

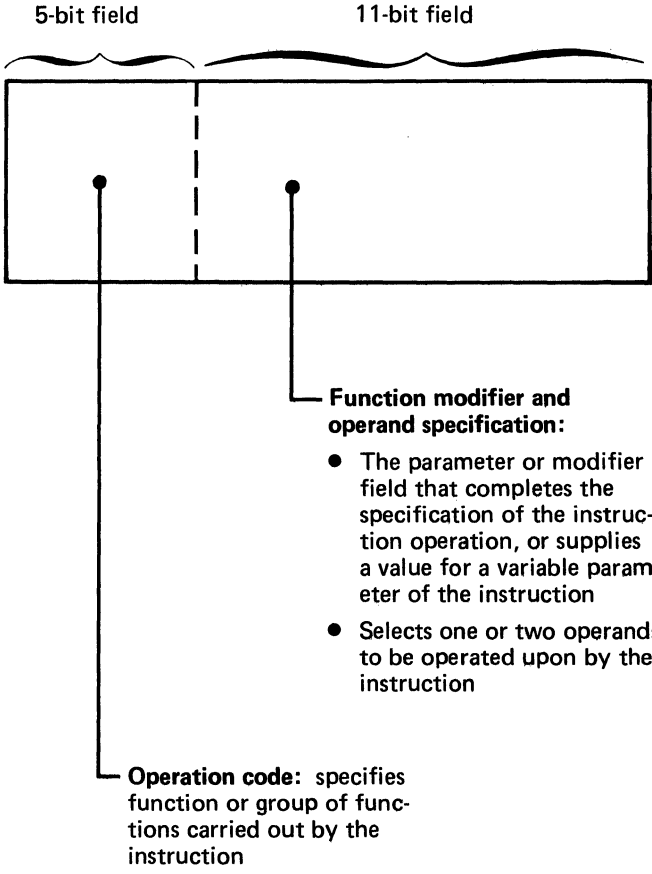
example, permits the programmer to reference data items by symbolic name. The assembler keeps track of data locations within a program and determines which registers have been set up for use as base registers. When a choice is available, the assembler determines whether to use direct addressing of the data item or base relative addressing, depending upon which instruction is shorter. Assembler support during program preparation time takes much of the decision-making burden away from the programmer.

Figure 77 shows the general format of those Series/1 instructions whose length is one word. All instructions, regardless of length, use the first five bits of the instruction as the operation code. In some cases, this code actually defines a group of operations. Each specific member of the group is identified in a field within the remaining eleven bits of the instruction. In other cases, the remaining eleven bits are sufficient to specify either the immediate data required by the instruction, the location of one or two operands, or a combination of both. All operations involving data in registers—or data whose addresses are in registers—can be encoded within the single-word instruction format shown in Figure 77.

Additional words are appended to the instruction format only if operands in storage are referenced using addressing modes that require data too long to fit within the single-word instruction. Figure 78 lists the addressing modes and their additional storage requirements. As noted in Chapter 4, base relative addressing modes are powerful tools in the design of systems; this factor more than compensates for the additional storage space required for their use.

Instructions Used for Data Movement

Application tasks are very much involved with the creation and use of data bases. Consequently, a significant portion of an application program consists of instructions that move data items of various types and lengths from one place to another in storage, or between storage and registers. Figure 79 lists the Series/1 instructions available for this purpose.



Most instructions can be written in a single-word format.

Operand specification includes register specification, immediate data, or data in storage referenced with one of the many addressing modes.

Some addressing modes require additional information which leads to two- and, sometimes, three-word instructions.

Figure 77. The basic one-word instruction format

(r)	:	Address of the operand is in register r.	No additional instruction word needed for these addressing modes
r	:	Operand itself is in register r.	
(r)*	:	Register r contains an address of a storage location in which the actual address of the operand is stored.	
(r)+	:	Address of the operand is in register r. Increment the contents of register r after using it.	
(r,d)	:	Add the displacement d to the contents of register r to get the address of the operand.	One additional instruction word needed for these addressing modes to contain the full address or displacement
(r,d)*	:	Add the displacement d to the contents of register r to get the storage address where the operand address is stored.	
d_2 (r,d_1)*	:	Add the displacement d_1 to the contents of register r to form an indirect address; add the displacement d_2 to the contents of that storage location to determine the operand address.	

Some addressing modes require that an additional word be added to the instruction to contain the data required to calculate the effective storage address.

Figure 78. Addressing modes and their additional storage requirements

Basic Data Movement Instructions

Most data items are a byte, word, or doubleword in length and are widely dispersed in application programs. These items may also reside either in registers or in storage requiring that they often be moved to and from these areas. As a result, the system provides instructions for moving each data type; these instructions support movement from registers to storage, storage to registers, and storage to storage for byte, word, or doubleword data items. Instructions with more specialized use, like Move Byte Immediate and Move Byte and Zero, are less versatile (the former moves the byte to a register only and the latter from storage to a register only). This restriction occurs because of the internal constraints on available instruction formats and the external conventions of their use. Register to register data moves are, of course, limited to full words—the length of the registers.

Floating-Point Data Movement Instructions

It was indicated in Chapter 3 that the format of floating-point numbers is identical to that in the IBM System/370 computers and, as a consequence, that conversion from integer to floating-point formats was a simple procedure. The power of the microprocessor on the floating-point optional processor is such that a user can realize additional savings in this conversion process. The Floating Move and Floating Move Double instructions permit floating-point numbers to move between floating-point register pairs, and—in both directions—between floating-point registers and storage. Because the data must move through the microprocessor, the system provides additional instructions to convert from integer to floating point or floating point to integer during the move. These instructions, Floating Move and Convert, and Floating Move and Convert Double, further simplify the conversion between number representations. As long as all conversions and operations are performed with the floating-point instruction set, normalization problems do not occur. Users must be aware, however, that the instruction processor assumes floating-point numbers in storage are

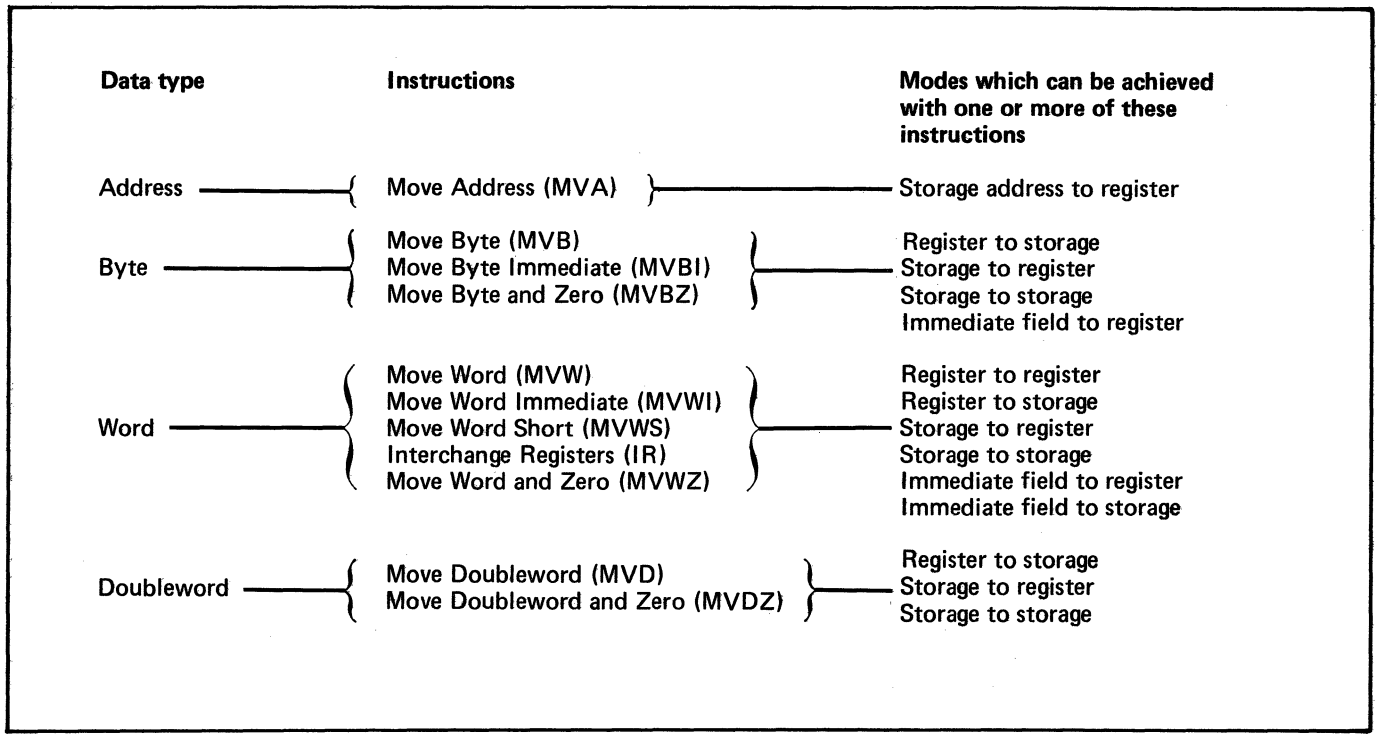


Figure 79. Series/1 instructions and modes for data movement (1 of 2)

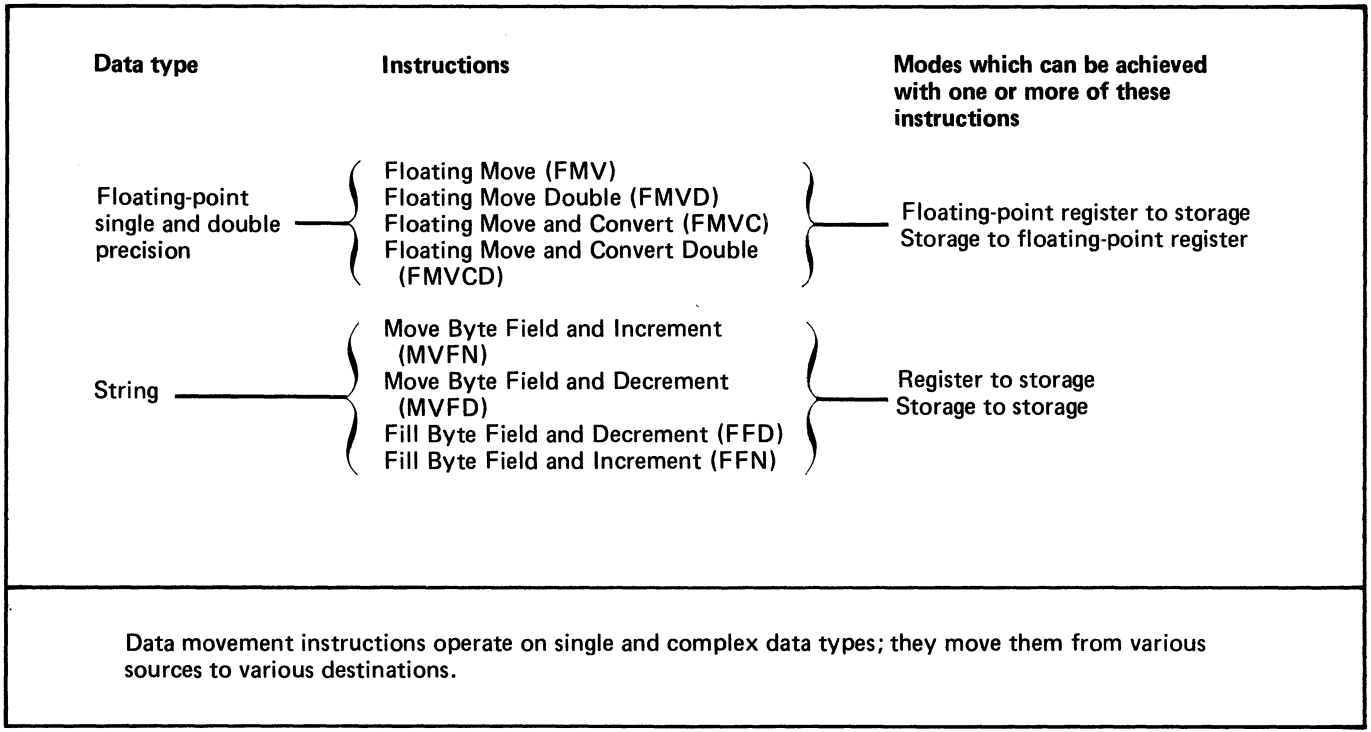


Figure 79. Series/1 instructions and modes for data movement (2 of 2)

already normalized. If the numbers are not normalized, errors may occur during the computations.

String-Data Movement Instructions

String-data movement involves moving one string in storage into another location or using a character stored in a register to initialize a storage string. In either case, movement is one byte at a time; the system stores the string's addresses in registers and modifies them each time a byte is moved. The two variations, increment and decrement, determine the end of the string from which the movement begins. At the initiation of the movement, the address of the first byte to be moved is in a register and changes to point to the next successive byte in the string. Register seven contains the number of bytes in the string and is decremented each time a byte moves. Consequently, the system may interrupt these instructions between any pair of successive byte-moves without destroying data. The instruction is simply restarted using the information currently in the registers. As mentioned earlier, very long string-data moves can occur without impacting the ability of higher-priority tasks to respond to interrupts.

Special Data-Type Movement Instructions

Because applications often reference separate data areas dynamically, addresses are as common a data type as are the more conventional numerical and character data types. The Move Address instruction permits the calculation of effective addresses and their loading into registers to perform:

- Base relative addressing
- Indirect addressing
- Building tables of pointers to data areas and routines
- Allocation of internal buffers
- Similar applications

The system includes other instructions because:

1. They are frequently used in practical operations
2. Their elimination would require replacement with awkward sequences of instructions

Examples of these instructions are: Interchange Registers, and Move Byte and Zero. These instructions are of special interest because they perform two operations concurrently; they can be used to control concurrent access to critical task sections, data areas, and other situations that arise in the implementation of applications by a set of cooperating, parallel-executing tasks. They are discussed further, from this point of view, in the section of this chapter entitled "Instructions Associated with Structured Programming and Control of Concurrency."

Instructions Used for Arithmetic and Logical Operations

Just as data types are moved between registers and storage, so are they also operated upon arithmetically and logically. Figure 80 shows the various data types and the arithmetic operations the system can perform on them. Arithmetic operations involve combining two operands (by addition, subtraction, multiplication, or division) to produce a result. The Series/1 instructions generally put the result in the place of the second operand. Hence a storage to register addition adds the contents of the storage location to the contents of the register and then puts the result in the register. The opposite is also true: a register to storage addition puts the result in the storage location without affecting the register contents. As Figure 80 indicates, to obtain the required operands and results, the Series/1 supports addition and subtraction of various data types for almost all combinations of registers and storage locations.

Storage to storage arithmetic operations—especially when they involve doubleword operands—are interesting from the point of view of program efficiency. Each doubleword operand stored in registers occupies two successive registers; only the instruction in the lower-numbered register references the operand. Because only eight registers are available to the user, applications may involve contention for register use: for instance, registers may be simultaneously needed as base registers and for temporary storage. Performing the

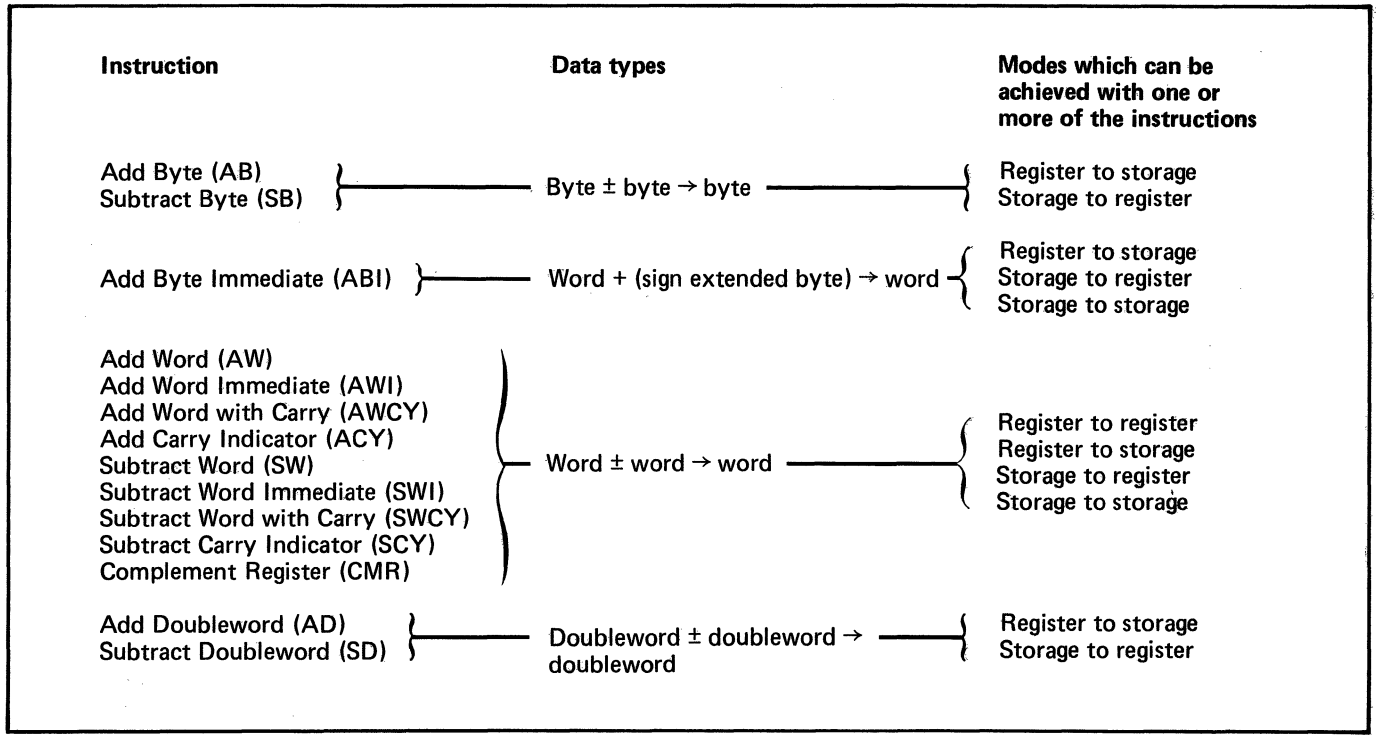


Figure 80. Arithmetic operations, data types, and modes (1 of 2)

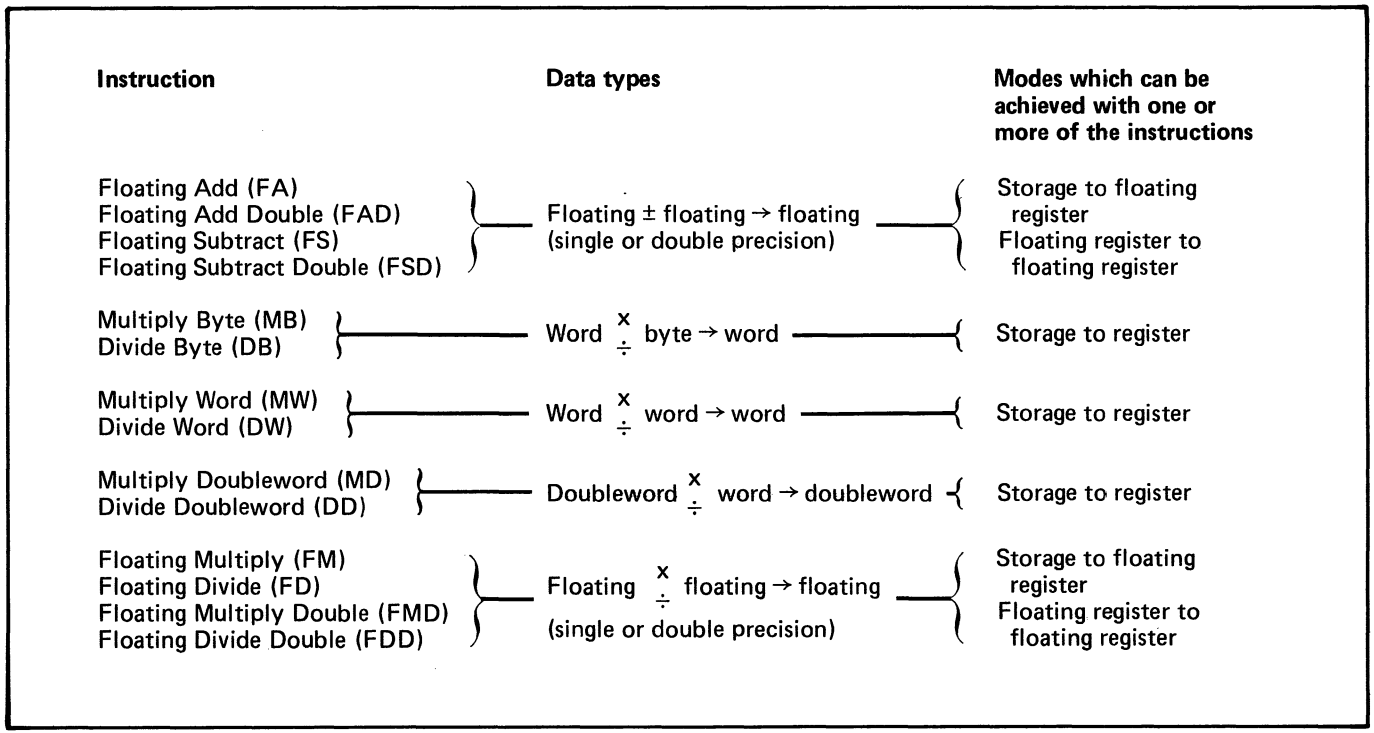


Figure 80. Arithmetic operations, data types, and modes (2 of 2)

arithmetic operations storage to storage does *not* require saving and restoring registers frequently throughout a program. Storage to storage operations execute faster than the procedure of bringing the data into registers, performing the operation, and then restoring the result to storage.

Numeric Data Operations

Multiplication and division assume signed numbers only, as mentioned in Chapter 3; but the system has designed these functions to accommodate combinations of different length operands—a situation that frequently occurs in actual practice. However, multiplication of a 16-bit operand by an 8-bit or 16-bit operand yields either a 24- or a 32-bit result; it is then necessary to store this result in a pair of registers and manipulate them to produce the desired, single word result.

The Series/1 Multiply Word instruction multiplies a word by another word and expects the result to be representable by a word. If the product is too large, overflow exists and is so noted in the level status register. To retrieve the larger product, the user first converts the word to a doubleword by sign extension and then employs the Multiply Doubleword instruction. Similarly, the Multiply Byte instruction multiplies a register contents (one word) by the value of a byte in storage and produces a single word result which replaces the register contents. The user chooses specific multiply and divide instructions based on observation of the most often used, practical combinations.

Arithmetic operations on floating-point numbers pose a different set of problems in the design of a system. Frequently, applications which use floating-point numbers involve exceptional situations where either an overflow or underflow condition invalidates the computation. Rather than require the testing of indicators in the level status register (as discussed earlier for the arithmetic operations on integers), the system generates an interrupt when the result of an operation overflows, underflows, or an attempt to divide by zero occurs. Through the internal interrupt mechanism of the Series/1, applications using

these operations do not have to test repeatedly for exceptional conditions—instead, the conditions are treated as events and the system interrupts only when they occur. The implementation of this feature is expedited by using an input/output slot to house the floating-point processor and its internal microprocessor.

Floating-Point Data Operations

The floating-point instructions shown in Figure 80 provide all four arithmetic operations for both single-precision (32 bits) and double-precision (64 bit) data formats. Furthermore, the instructions permit storage to floating-point register or floating-point register to floating-point register versions in all cases (recall that four 64-bit floating-point registers are implemented separately from the eight user registers on a per level basis) and are located in the optional floating-point hardware processor. All instructions refer to these registers and not the standard registers in the level status block.

Logical operations on data items commonly occur in applications to:

- Isolate bit fields
- Manipulate flags and condition indicators
- Perform other detailed computations where individual bits must be manipulated

Standard operations are AND, OR, Exclusive OR, and Reset. Some processors perform these operations on an individual bit basis; the instruction addresses the word and a field within the instruction addresses the particular bit of interest. This procedure is especially desirable when the system tests the status of a particular bit—a very common operation. The procedure is less desirable for the previously discussed logical operations because it is a common practice to store multiple similar flags, bits, or bit fields within a word; to perform the operations on a whole word is, then, an efficient operating procedure.

Logical Data Operations

The Series/1 processor logical instructions combine features of both procedures. The logical operations AND,

OR, Exclusive OR, and Reset are performed on multiple bits (in bytes, words, or doublewords). The testing instructions operate on a specific bit within a data type. This choice, again, is made after analyzing common usage procedures in application and system programs. Figure 81 shows the logical instruction set (except for the testing-bit instructions which are discussed in this chapter under the heading "Instructions Associated with Testing Operations' and Computations' Status"). Notice that the OR and Exclusive OR instructions operate on bit fields of byte, word, and doubleword lengths. Furthermore, all instructions permit both register to storage and storage to register modes.

The Reset bit shown in Figure 81 operates on bytes, words, and doublewords with the same choice of register to register, register to storage, and storage to register modes as the OR and Exclusive OR groups. These instructions reset those bits in the second operand which correspond to one-bits in the first operand. Other bits in the second operand are left unchanged. This selective resetting of bits is useful in manipulation of flags and indicators. However, it should be noted that this operation is equivalent to an AND operation with the first operand negated before the operation. Hence, to get a complete set of AND operations to parallel the other logical operations, the user need only negate the first operand. This is a straightforward operation in typical AND applications like masking. The instruction Invert Register also accomplishes the negation if it cannot be done at program preparation time.

Shifting Data Operations

Shifting instructions perform equally powerful bit manipulation. The system can shift the contents of registers and pairs of registers to the left or right and fill the vacated positions in three different ways: 1) by circular shifts; 2) by logical shifts; and 3) by arithmetic shifts. Figure 82 illustrates the options. Circular shifts take bits shifted out of one side of the register and insert them in the other. Logical shifts (right and left) replace vacated register bit positions with zero.

Arithmetic shifts to the right propagate the sign position into vacated register bits. Through combinations of single and doubleword length operations, these instructions effectively pack fields into words.

One characteristic of these instructions' format is that the user can specify the number of shifted bits that are to be coded into the instruction or to be placed in a register. This capability—combined with the carry and overflow indicators in the case of left-logical shifts—provides information about the bits shifted out of the word. Specifically: the system loads the last bit shifted out of the left side of the register or register pair into the carry indicator. The overflow indicator signals when the shift changes bit zero (the most significant bit—the sign bit). These indicators are very useful in programming multiple-precision, arithmetic operations.

A common procedure in small computer applications involves using words to contain flags, and testing to determine which flag has changed. Instead of individually examining each successive bit, it is more convenient for the system to set up a word which contains a one in the bit position of interest and zeros elsewhere. The Series/1 provides a special instruction, Shift Left and Test, which shifts left until a non-zero bit shifts out of the register. The maximum length of the shift is preloaded into another register. When the one bit shifts out of the register being tested, shifting stops; the register containing the shift count then contains the number of bits remaining to be shifted. This translates immediately to the bit position of the non-zero bit. In this manner, the system can examine multiple bits with one instruction.

In summary, the Series/1 offers a variety of instructions to manipulate a variety of data types. Together these instructions provide the user with the basis for coding critical tasks while retaining control over those exceptional conditions which might arise for special data values. The testing capability for these conditions is present in the level status register condition bits which reflect the result of arithmetic and logical operations. The appropriate processor manual describes, for each instruction, the specific effects of exceptional conditions.

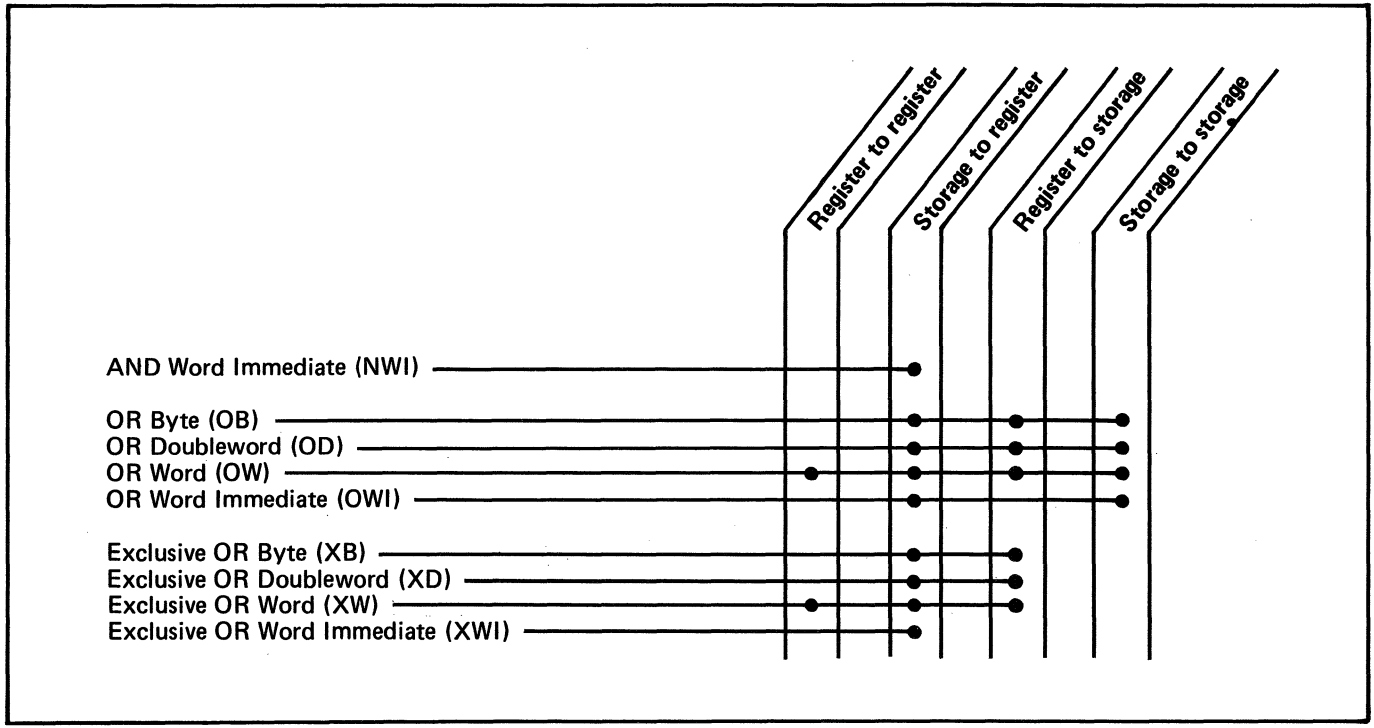


Figure 81. Logical instruction set and modes of use (1 of 2)

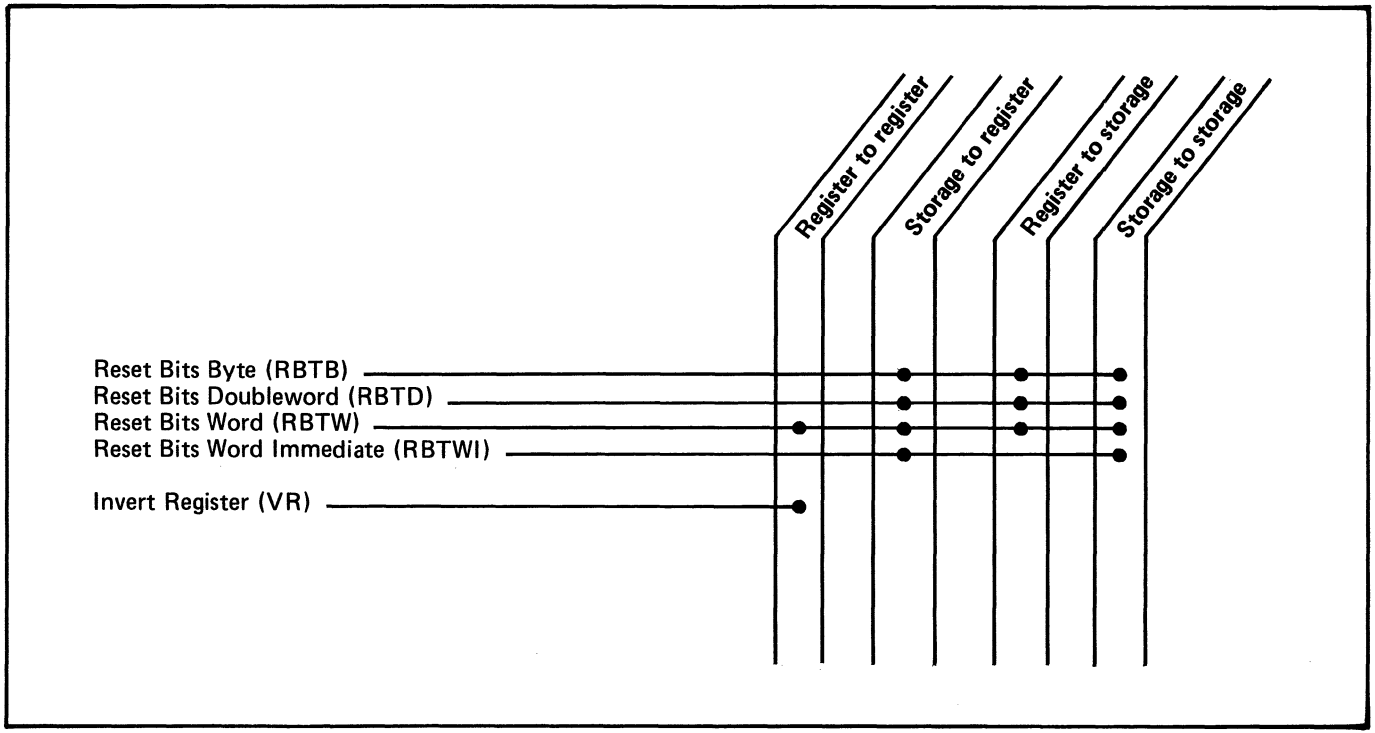
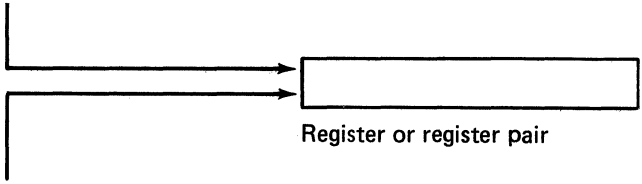


Figure 81. Logical instruction set and modes of use (2 of 2)

Shift right—arithmetic or logical

Shift Right Arithmetic (SRA)
Shift Right Arithmetic Double (SRAD)
Shift Right Logical (SRL)
Shift Right Logical Double (SRLD)

Replace vacated bits by zeros when shifting right *logically.*

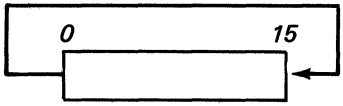


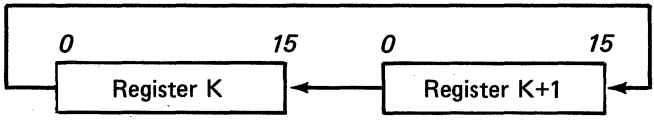
Register or register pair

Replace vacated bits by original sign bit if shifting right *arithmetically.*

Shifting left—circular

Shift Left Circular (SLC)
Shift Left Circular Double (SLCD)





Register pair shifting left

Figure 82. Options for shifting register contents (1 of 2)

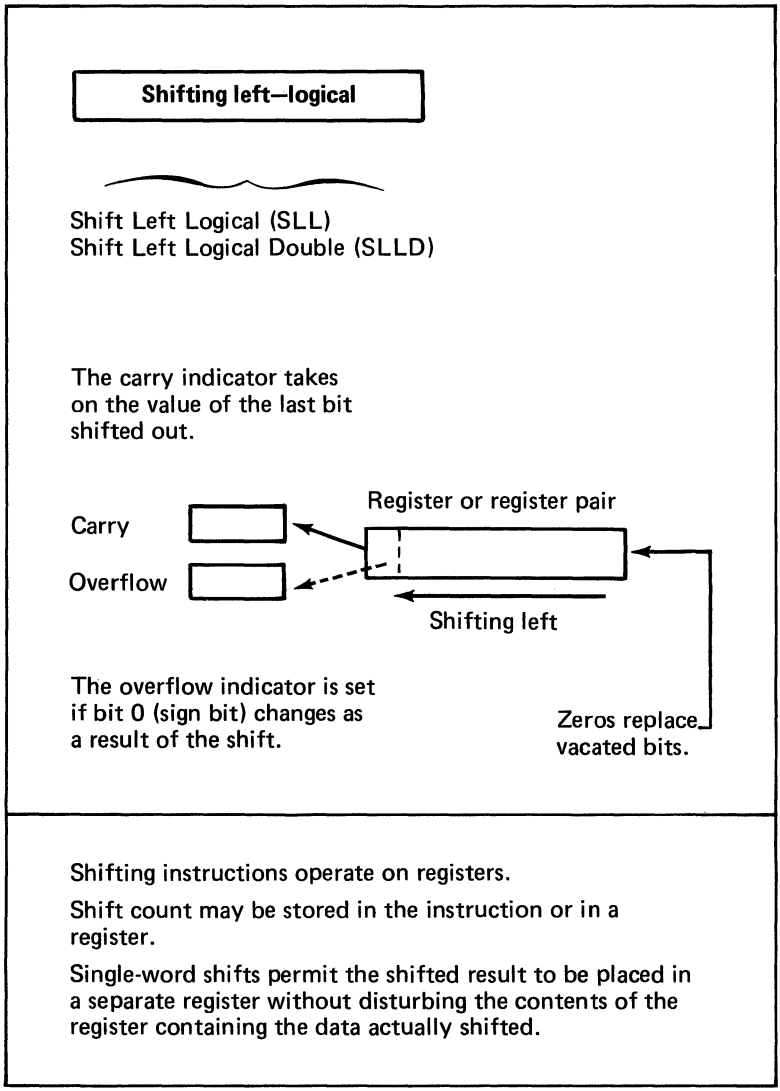


Figure 82. Options for shifting register contents (2 of 2)

Instructions Associated with Testing Operations' and Computations' Status

Decision making is a major characteristic of computer applications. This process involves testing:

- Variables
- Results of computations
- Results of processor operations; and then performing different instruction sequences depending upon the tests' results

As shown in Figure 83, the process occurs in two steps. First, the system sets an indicator either by executing a special instruction which makes the desired test, or as a by-product of some other operation. For example, every arithmetic operation leaves indicators concerning overflow, carry, negative result, zero result, and even result in the level status register. Every input/output operation also leaves a status code in the level status register. Secondly, the system tests the condition flags via Branch or Jump instructions to permit the transfer of control to the appropriate location shown by the setting of the indicators. Notice that indicators are duplicated in level status registers for each level of hardware priority; consequently, the conditions will not be inadvertently changed should a higher priority interrupt occur before the indicators have been tested.

Interruptible and Non-Interruptible Testing Instructions

Specific instructions for testing indicators are also shown in Figure 83. The Compare instructions reference two bytes, words, doublewords, or byte fields. Comparison is equivalent to subtracting the two operands with the indicators set to correspond to the result (zero result means the operands were equal). In addition to a comparison of these simple data types, a string of bytes can be compared to a register byte or another string of bytes. A string is specified by: 1) the address of the corresponding byte in each string (addresses are stored in two registers); and 2) the number of bytes in register seven (Figure 84). The Compare Byte Field instructions perform a byte by byte comparison; after each comparison, the instructions decrement the count in register seven and increment or decrement the addresses in the two address registers. The test proceeds byte by byte until the comparison asked for in the instruction

is found to be true for a specific byte (equal or not equal depending upon the instruction selected). At the conclusion of the test:

- Register seven contains the number of bytes not compared (or zero)
- The two registers point to the first uncompared byte-pair, *or*
- If all the bytes are compared, the two registers point to just beyond the last byte

Notice the system design inherent in this instruction: because each byte comparison updates registers, the system can interrupt the instructions between comparisons and restart them with no loss of data. Once again, this ability permits higher-priority levels to respond quickly even when the system is comparing lengthy byte strings. The comparing procedure could be time consuming and cause delays if the instruction had to be executed without allowing interrupts.

Bit and Field Testing Instructions

In addition to testing or comparing bytes, words, doublewords, and strings, the system can test individual bits. The Test Bit instruction addresses the desired bit as a displacement—limited to 63—from the beginning of a byte. With this instruction, any bit in an eight-byte data type can be individually addressed and tested. The other test bit instructions in the Series/1 processor permit combinations of testing and setting, inverting, or resetting. These combinations are very useful in controlling concurrency. The section of this chapter entitled "Instructions Associated with Structured Programming and Control of Concurrency" discusses them.

The Test Word Immediate instruction provides a masked testing of a combination of bits within a word. The instruction tests only those bits in the mask (the immediate word) in the addressed word. Indicators are set depending upon whether all the bits to be tested are zero, one, or a combination of zeros and ones. With this instruction, combinations

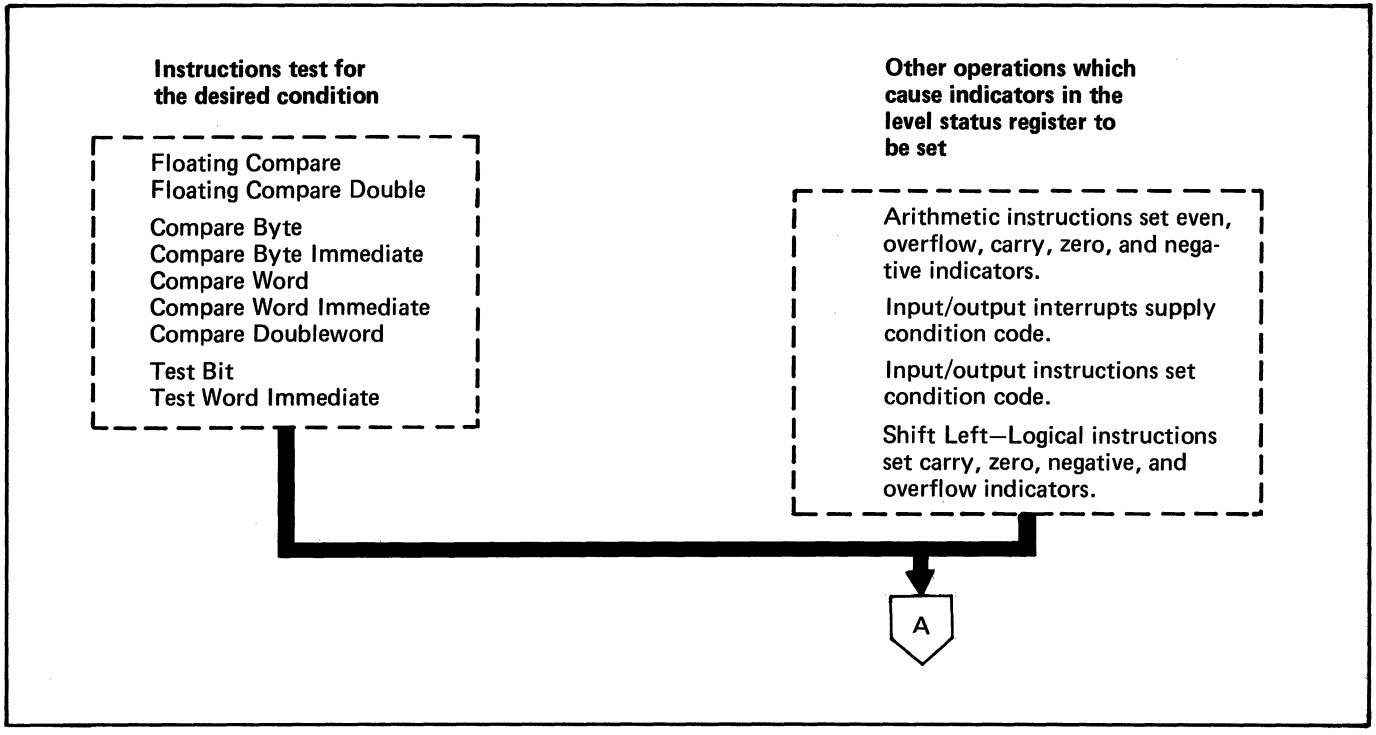


Figure 83. Operation and computation testing instructions (1 of 3)

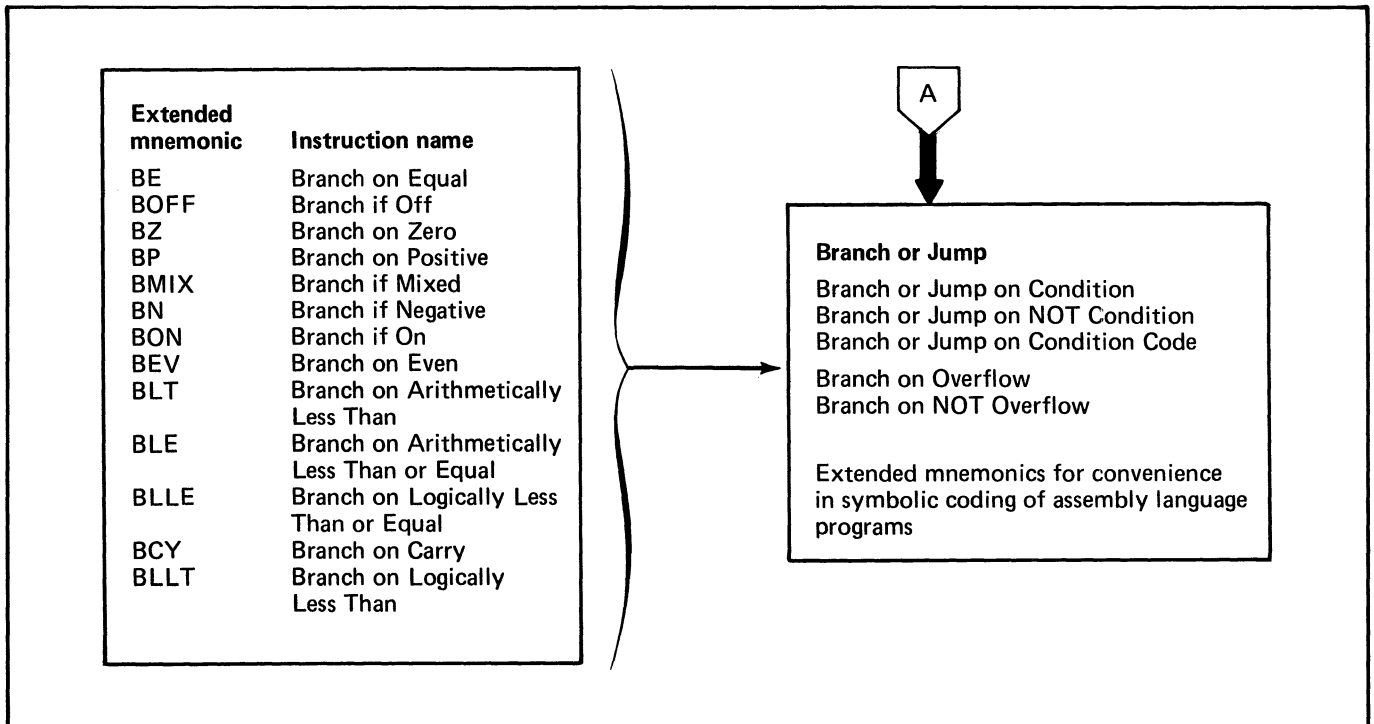


Figure 83. Operation and computation testing instructions (2 of 3)

Instructions which change the sequence of program execution are dependent upon the setting of indicators in the level status register.

Control over the execution sequence:

1. Depends upon the results of previous operations or actions
2. Uses instructions which set conditions in the level status register or Test or Compare instructions. These instructions test specific items which are followed by condition-dependent Branch or Jump instructions.

Branch: permits arbitrary addresses but requires a two-word instruction.
Jump: is relative to the instruction address register. It is limited to a range of plus or minus 128 words. The instruction is one word long.

Figure 83. Operation and computation testing instructions (3 of 3)

of flags can be efficiently tested to determine if a combination of conditions is true. This is a very practical capability. For example, a user might want to make an action dependent upon three conditions: 1) a motor running, 2) a valve being closed, and 3) a pressure reading greater than some preset value. Allowing one bit in a status word to indicate the state of each of these three logical conditions permits testing for the simultaneous occurrence of the conditions with a single, masked test instruction.

Conditional Transfer Instructions

Program sequence control is through Branch or Jump instructions; the former permits transfer of control to arbitrary locations; the latter permits transfer of control to locations within approximately 256 bytes of the Jump instruction. The Jump instruction is actually relative to the current value of the instruction address register and is only a single word long. Almost all Branch instructions have an identical Jump instruction; if the range of the jump is short, Jump is the instruction preferred.

The Branch On Condition instruction contains a three-bit field which is coded to contain one of the eight different conditions. The assembler simplifies coding of these branches or jumps by providing different instruction names for each specific condition (Figure 83). The instructions can test each specific condition. In addition to these conditions, Branch and Jump instructions permit a branching operation that depends upon the value of the condition code written into the instruction. Recall that condition codes are reported in the level status register after every input/output instruction, and after every input/output interrupt that signifies the result of the operation or reason for interrupt.

The Series/1 provides one additional sequence control instruction, Jump On Count, for control of looping within programs. As shown in Figure 85, this instruction permits a loop to be iterated as many times as is indicated by an integer or count set up in a register. Each time through the loop, the Jump On Count instruction is executed. This instruction tests to see if the register is zero; if it is, the

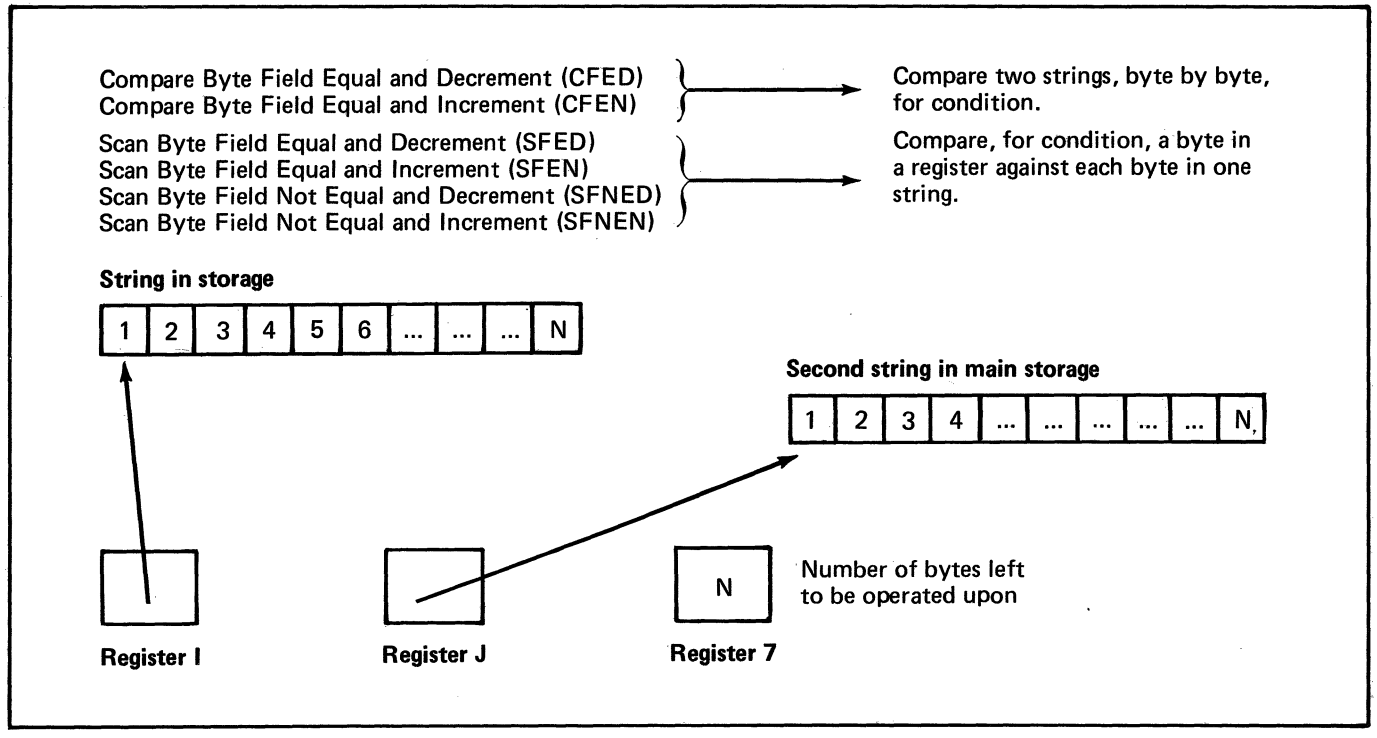


Figure 84. Comparing a string of bytes (1 of 2)

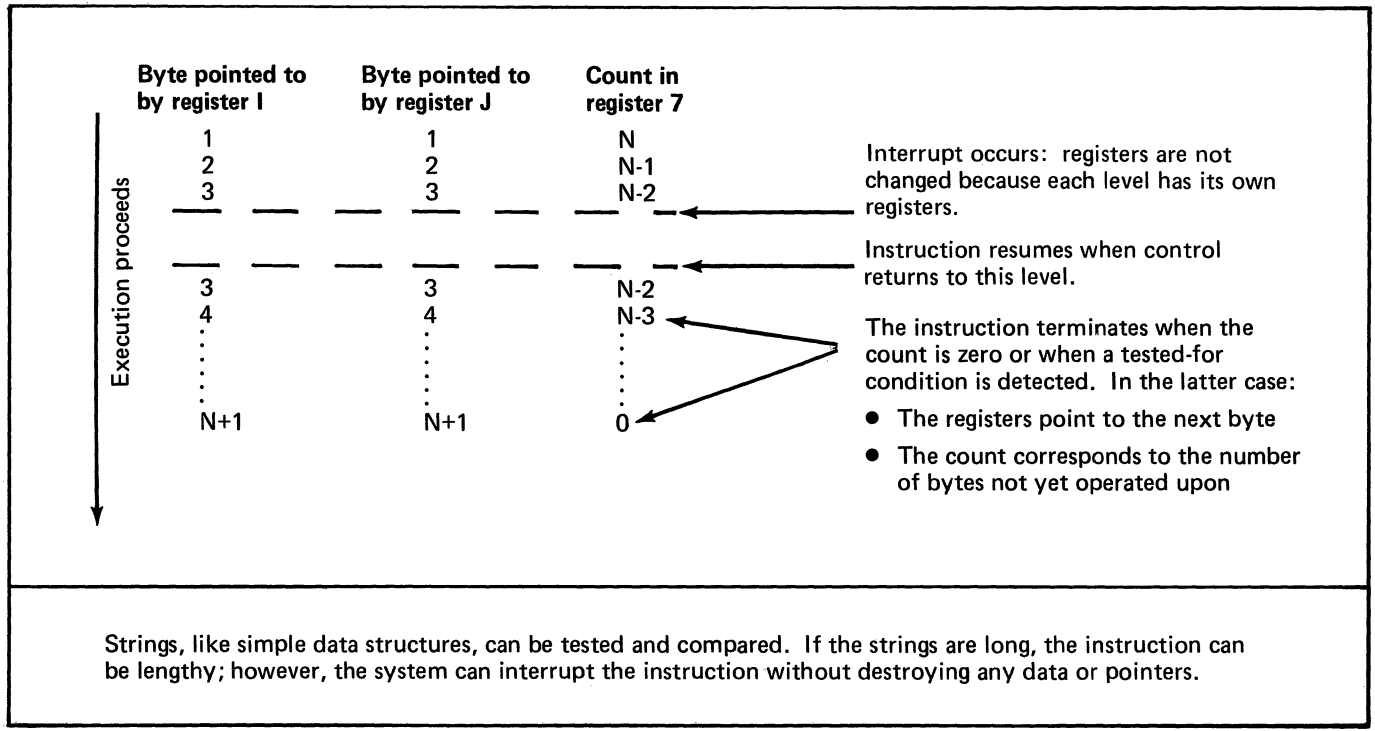


Figure 84. Comparing a string of bytes (2 of 2)

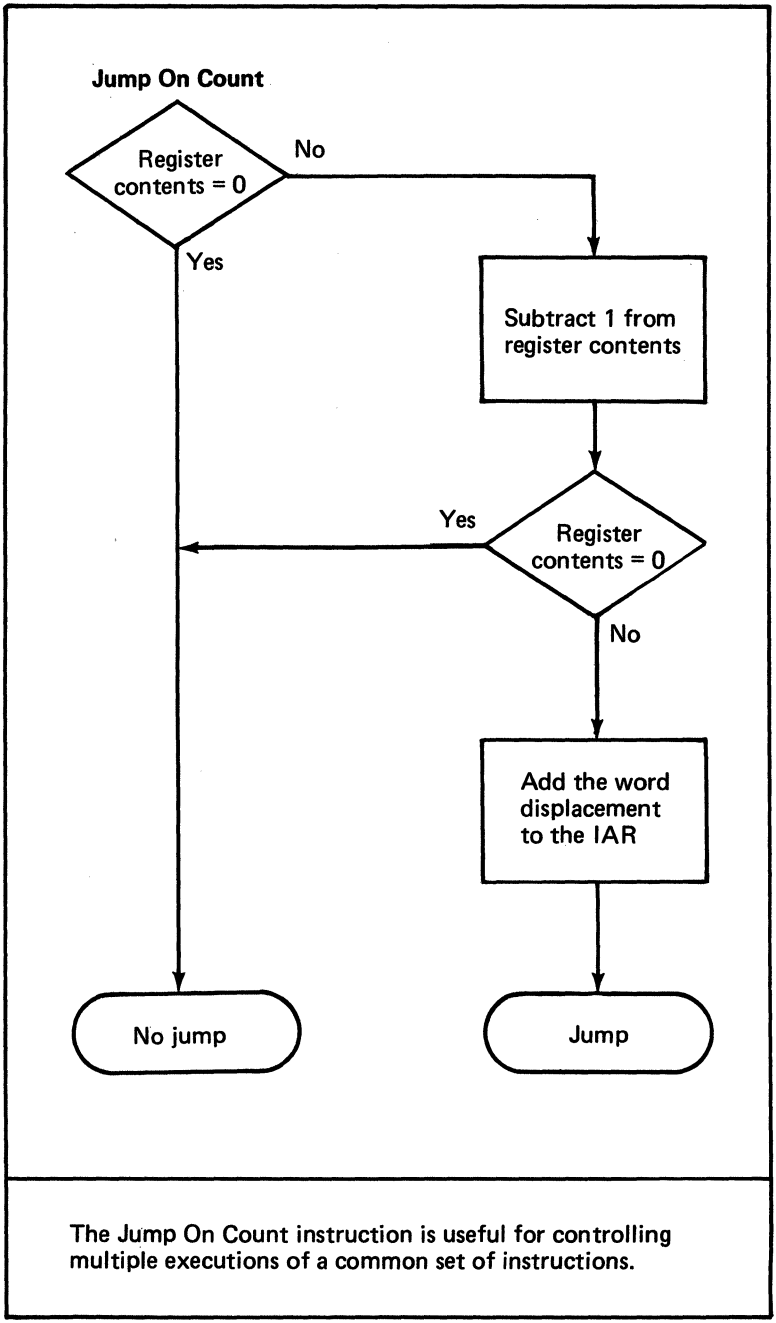


Figure 85. The Jump On Count instruction (1 of 2)

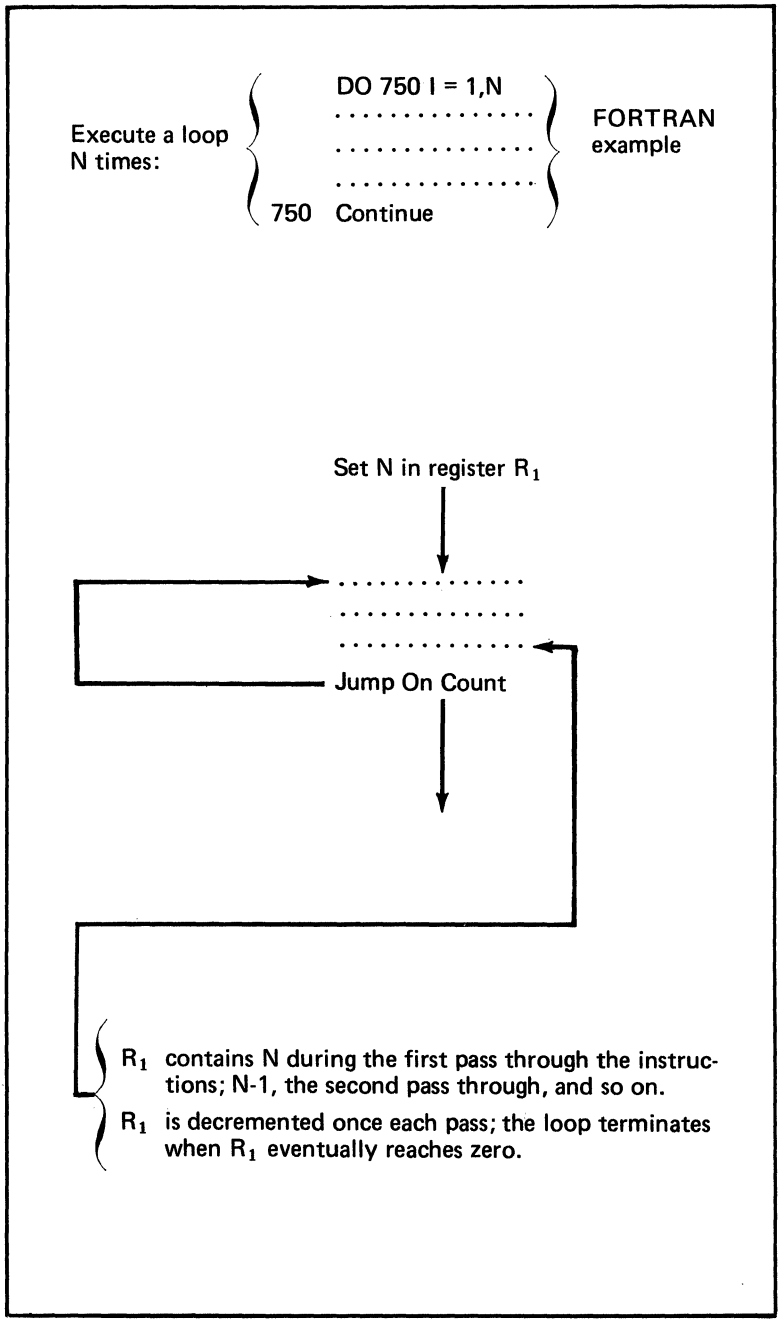


Figure 85. The Jump On Count instruction (2 of 2)

instruction continues (exits from the loop). If it is not, the count is decremented and the register is tested again in the same way.

Instructions Associated with Structured Programming and Control of Concurrency

Good computational instruction sets cannot stand alone. Small computer applications require an environment consisting of a set of cooperating tasks; this environment implies structuring a large task into many separate tasks, and restructuring the tasks themselves into many small modules—which may be shared. The sharing of modules or data inevitably means contention when two modules attempt to use the same resource or update the same data item at the same time. These conflicts arise because multiple hardware priorities mean that tasks can interrupt other tasks at arbitrary points in time. The instruction set must permit both efficient structuring of programs and modules and control over concurrency. The Series/1 instruction set provides many alternatives for the solution of these contention problems, as shown in Figure 86.

Consider first, the problem of concurrency. As mentioned earlier, a user can often handle shared routines by making them reentrant. The strong set of Series/1 instructions that support stacks (Figure 86) was discussed in detail in Chapter 3. As demonstrated there, this set of instructions provides detailed solutions to the concurrency problem; this support need not be reiterated here.

Serializing Resource Usage

However, there are many situations in which the system cannot allow concurrency. In these cases, it is necessary to "serialize the use of the resource"; that is, to insure that only one task at a time uses the data, code, or other resource. Figure 87 shows the simplest way to handle this problem: disable interrupts upon entering a critical section or routine, and enable them again upon exiting. With interrupts disabled, no task switching can occur; as a result, no other task

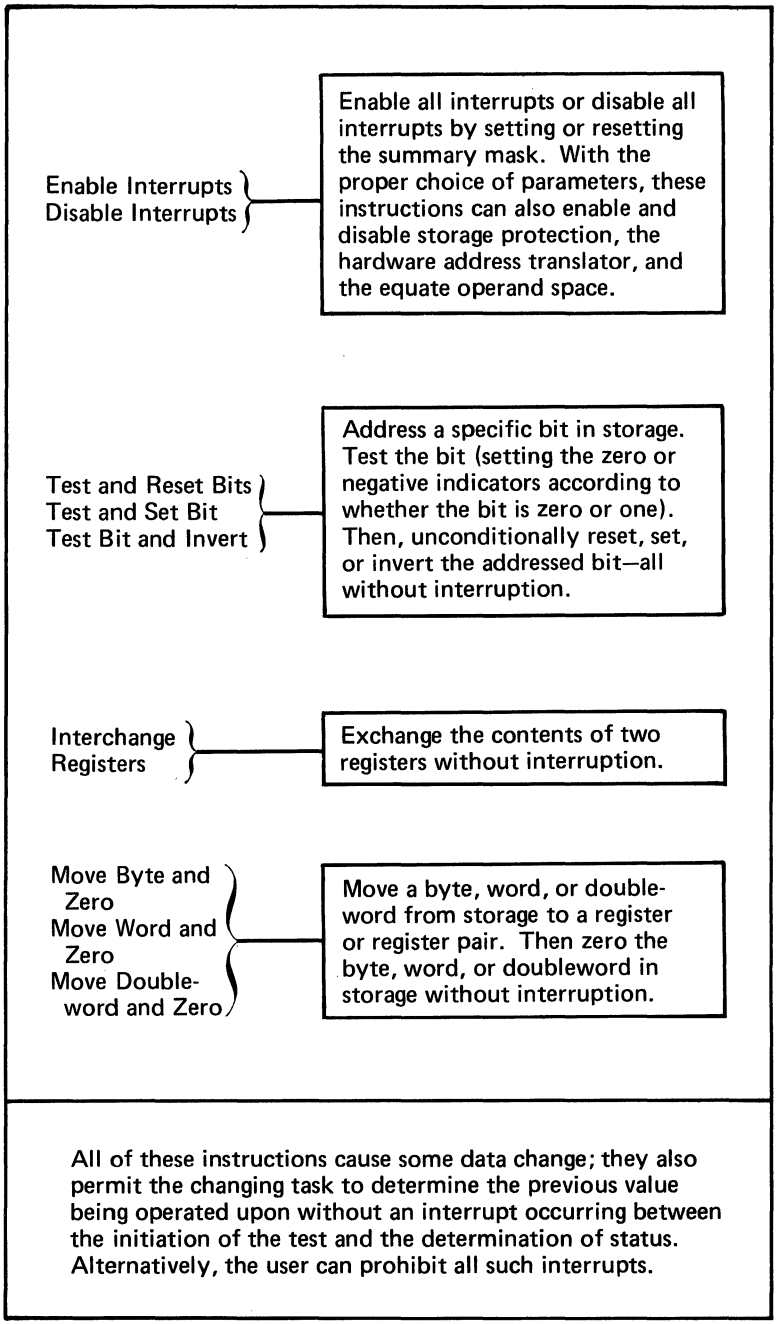
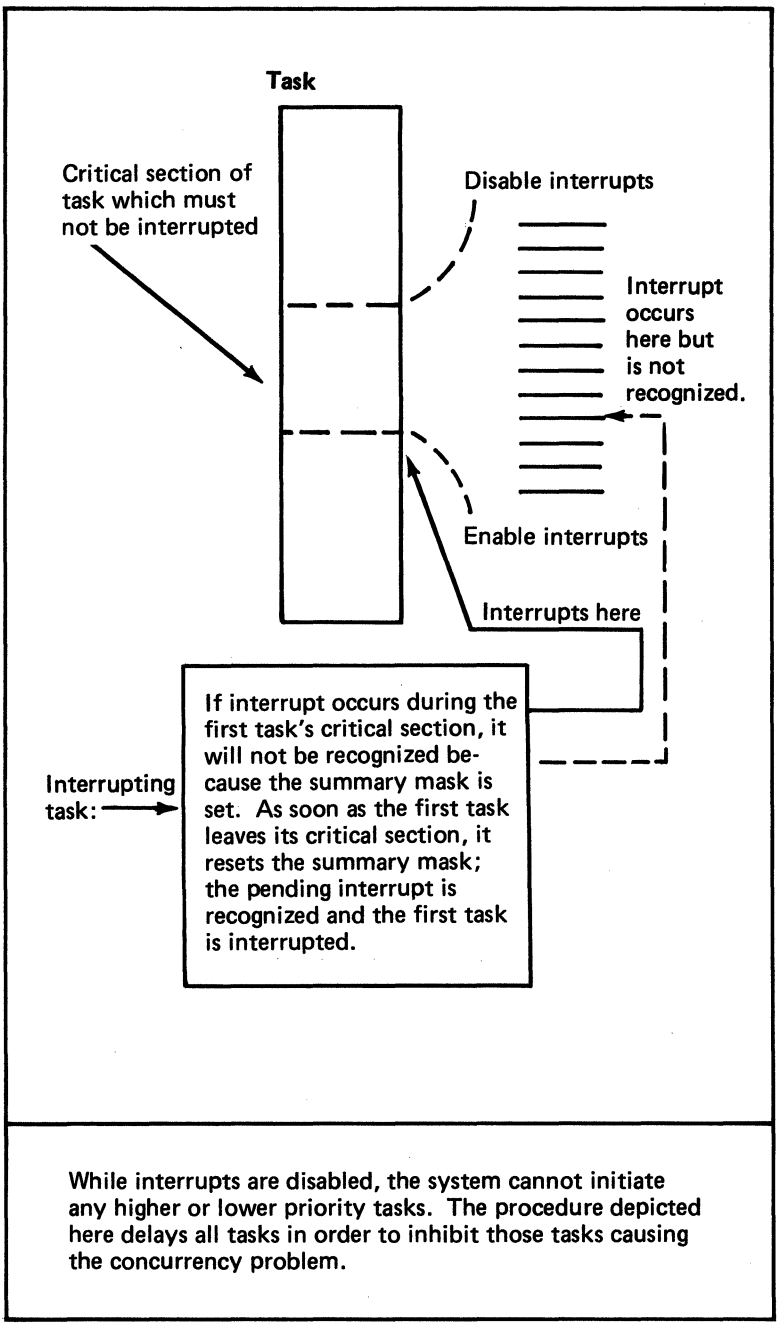


Figure 86. Instructions which can be used to control concurrency



While interrupts are disabled, the system cannot initiate any higher or lower priority tasks. The procedure depicted here delays all tasks in order to inhibit those tasks causing the concurrency problem.

Figure 87. Using disabling and enabling interrupts to control concurrency

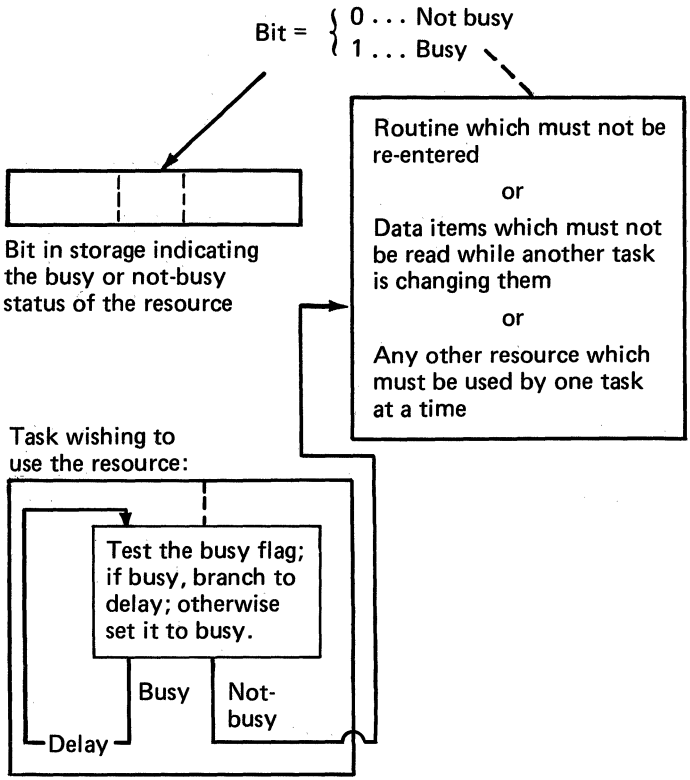
can use the resource until the current task is through with it. The disadvantage in this approach is that all other higher priority tasks are necessarily delayed during the time the interrupts are disabled. This solution, then, is practical only if the interrupt-disabling time is very short.

An alternative solution is shown in Figure 88. Here, a flag signals that a routine is busy; the system tests the flag before entering the critical section or routine. However, the status of the flag might change between the time it is tested and found not busy, and the time it is set to busy. This change can happen if a higher priority task interrupts between the operations at precisely the right moment. To prevent this from happening and to prevent the higher priority routine from reentering the critical section—where reentry causes an error—the system must test and set the busy flag in one uninterruptible operation.

Instead of disabling and enabling interrupts during the testing and setting of the flag, the Series/1 makes several instructions available to perform the job. As shown in Figure 88, Test and Set performs the two operations with a single uninterruptible instruction. If the flag is in one of the registers, the user can load the busy value into a second register, and then interchange the two tasks through a single instruction. This procedure is equivalent to the reading and testing of the flag that the Test and Set instruction performs. This procedure is not viable if the task attempting to reenter the routine is on a different priority level because each level has its own set of registers—the two tasks could not access on two different levels. The procedure is effective for tasks on the same level if the system switches control back and forth between them and the user does not want to relinquish some piece of code or data from one task to the other. The Move and Zero instructions may be used in a similar fashion because they also access the value of a data item and set its value to zero in one uninterruptible operation.

Application Software Modularizing

Structuring of tasks is the second important consideration affecting design of an instruction set. The Series/1



Use the Test and Set instructions which both test and set in *one uninterruptible instruction.* Testing the busy flag status and setting it to busy must be done without interruption. Otherwise, the task might test it and find it not busy, but be interrupted before it can set it to busy. The interrupting task might then seize the resource—because the task finds the resource not busy. If control returns to the first task, it would then consider the resource to be available—when, actually, it is not.

Notice in the Series/1 that each level has its own set of registers; consequently, tasks on different levels cannot use this technique.

Figure 88. Serializing the use of a resource using the Test and Set type of instruction

addressing modes facilitate this common characteristic of small computer applications. Separating data from tasks and sharing data among many tasks involves complicated addressing problems that were explained in the first chapter of this book.

Breaking up programs into modules requires that the system be able to:

- Branch to a module
- Execute the code in that module
- Then return—this is the subroutine concept

The Branch and Link, Branch and Link Short, and Jump and Link instructions provide this capability which is similar to the one used in the IBM System/370 computer systems (Figure 89). These instructions cause the system to store the address of the next instruction after the Branch and Link instruction in register seven. A branch then occurs to the address of the subroutine. Register seven then contains the return address. If the routine is reentrant, the routine may save the address on a stack. If the address is not needed during the execution of the routine, the routine may save it within itself or leave it in register seven. By common agreement, arguments passed to the subroutine may be passed through the registers. Alternatively, the argument may be placed in the calling program after the Branch and Link instruction. In the latter case, the system uses the address in register seven to access the parameters to be passed (Figure 90).

In this latter case—in order to skip over the data being passed—the system must change the address in register seven before the return occurs. The subroutine facility, in one form or another, is available in all higher level languages. The Series/1 Realtime Programming System also permits the user to pass a set of parameters from one task to another—a further enhancement of intertask communications.

One further technique which aids in structuring multi-task applications is the Series/1 ability to cause interrupts under program control. That is, one way to initiate a separate

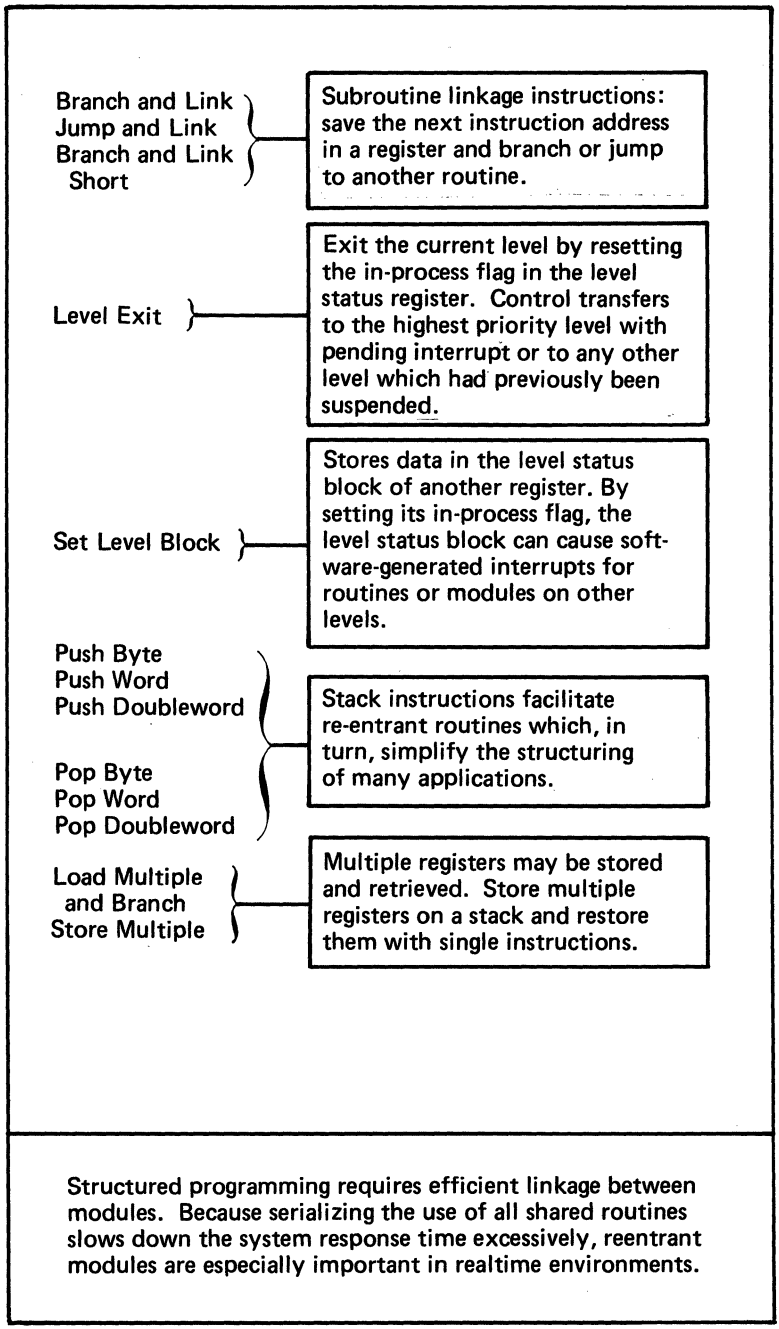
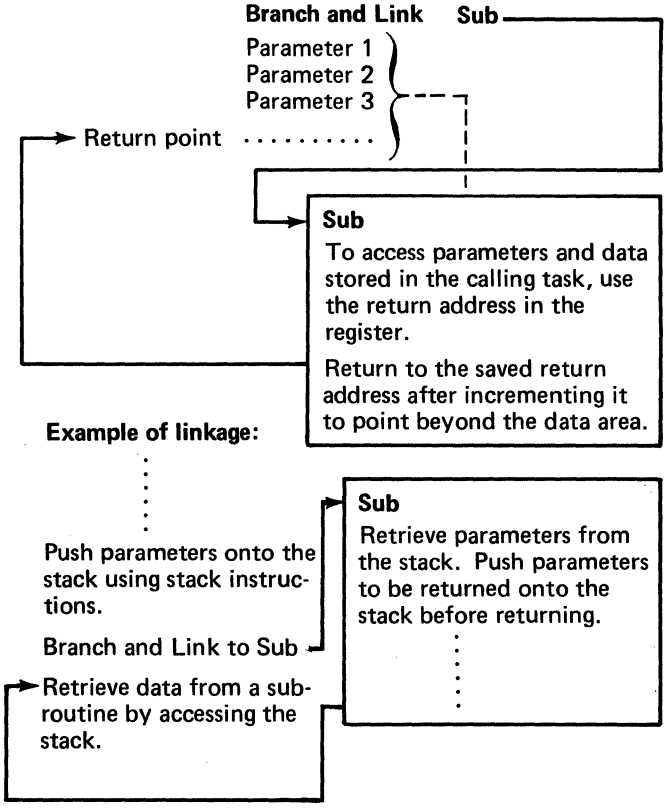


Figure 89. The subroutine concept

Structuring a task or program into modules is practical only if there is a mechanism for efficient linkage and transfer of the shared data.



Communication between a calling program and a subroutine may be done by:

- Using parameter passing techniques
- Using common areas of main storage
- Passing addresses of data buffers or stacks

Figure 90. Structuring a task or program into modules

module of a structured design is to attach it to an interrupt and then, by executing an instruction, cause the interrupt to occur in a second module.

The Set Level Block instruction is a privileged instruction which loads, for a specified level, a level status block from an arbitrary location in storage (Figures 89 and 91). This instruction is advantageous for the following reason. The level status register within the level status block contains an in-process flag which the program can pre-set to activate that level as soon as: 1) its level status block is loaded; and 2) no higher priority level is pending. By setting the in-process flag to "on" for a higher level, control immediately transfers to that level and the current level is set to pending. When the in-process flag is set to "on" for a lower level, it becomes active only when the current level exits by a Level Exit instruction which resets the current level's in-process flag. Thus, interrupts on higher or lower priority levels can be initiated under (privileged) instruction control.

The combination of task sets, partitions for storage management, subroutines which may be reentrant, and control over concurrency via the instructions listed in Figure 86 permits response implementations to typical, small computer application needs.

Instructions Associated with Management of the Processor

To control the overall processor, the Series/1 provides a set of privileged instructions that read and write those registers and variables not available to application tasks. The existence of privileged instructions is justified by the important need to maintain control over system integrity. That is, an application is usually realized as a set of cooperating tasks—with the cooperation consisting of shared resources, routines, data areas, and operating system facilities.

To preserve overall system integrity, the tasks must not—in general—interfere with one another. Consequently, any instruction that requires the resources or operation of

another task must be privileged so that any attempt to execute that instruction—even inadvertently—will cause a loss of control by the executing task. When not in the supervisor state, an attempt to execute any privileged instruction by an application task causes an exception interrupt to occur, and returns control to an operating system which determines the cause and takes appropriate action. This arrangement facilitates input/output:

- By using central routines
- By using the control operating system resources—via the Supervisor Call instruction (SVC) which causes an exception interrupt

In the same consistent manner, the system handles storage protect violations, referencing of mapped storage not available to the task, and similar errors.

Figure 92 lists the Copy and Set instructions which permit both the setting up and the accessing of data in registers not available to users. These instructions involve:

- Address key registers which control storage protection and storage mapping
- Segmentation registers
- The interrupt mask register
- Various level status indicators
- Similar indicators and registers which control execution of tasks and relations among tasks

The user must first understand the interrupt mechanism, the storage protection mechanism, and the storage mapping scheme. When the user becomes familiar with these, and similar functions, implementation of the Copy and Set instructions by an operating system—to initialize storage, set up segmentation registers, and other uses—becomes conceptually self-evident. Users should reference the appropriate processor manual to determine addressing modes, exceptional conditions, and other specific information for each instruction.

Most applications need these instructions only as an assurance that they are present to enable the operating system to

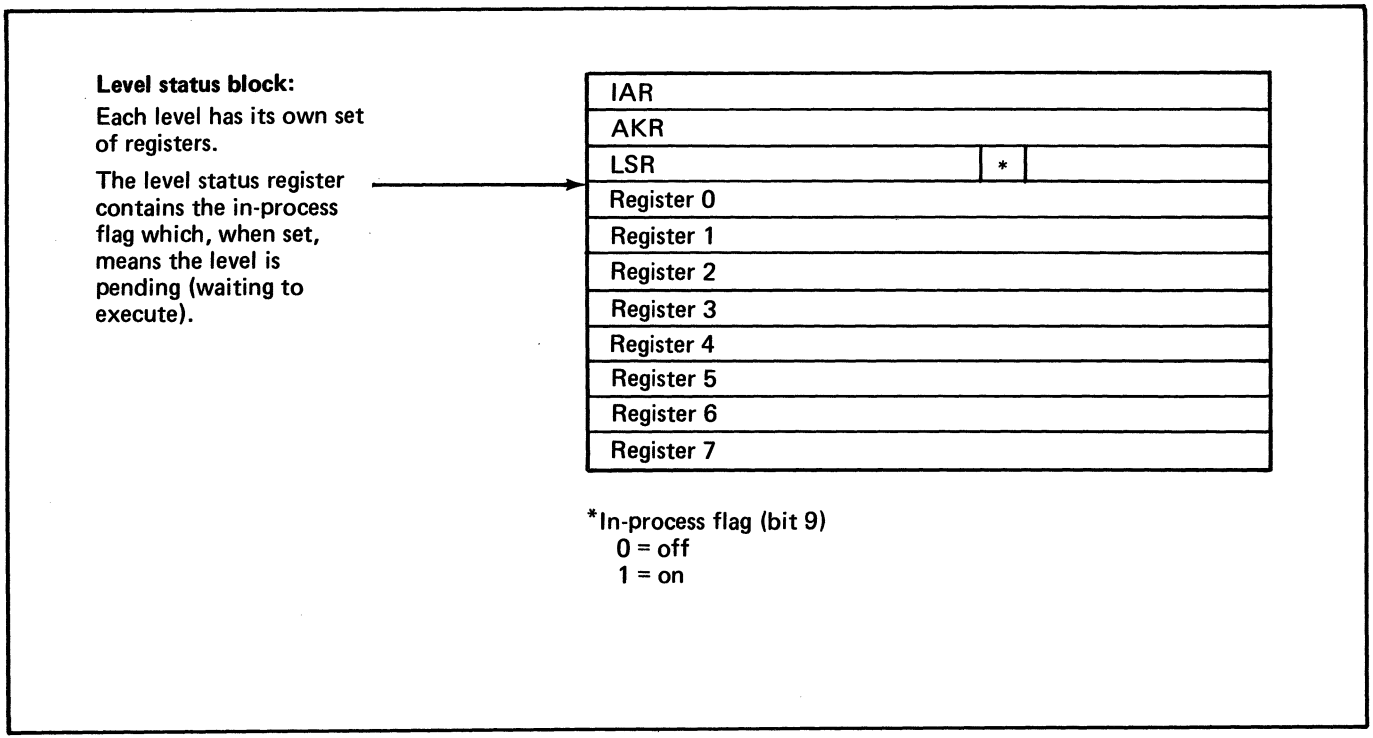
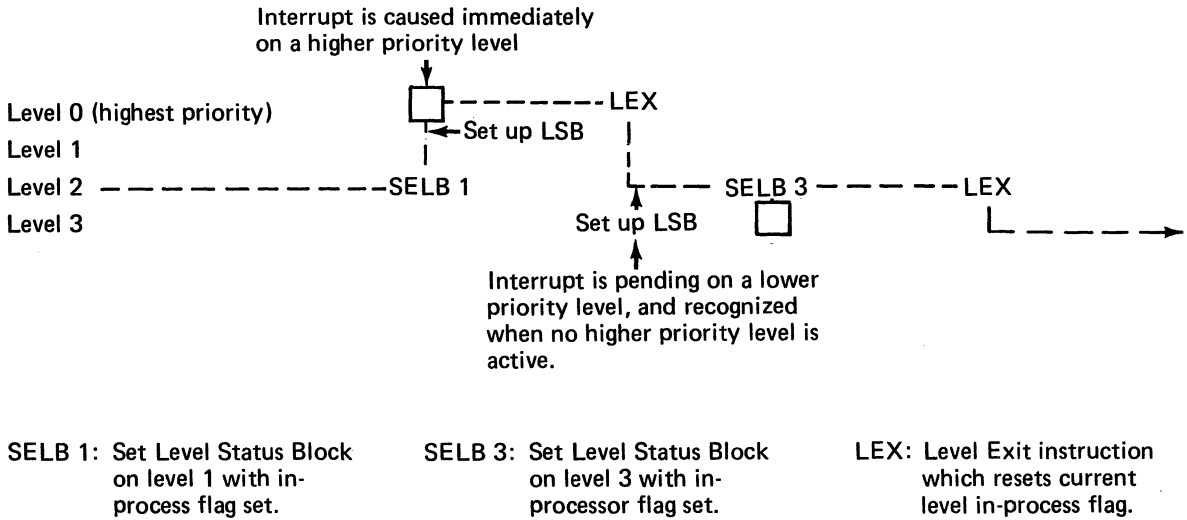


Figure 91. The level status block and module scheduling (1 of 2)



Modules in an application can cause interrupts and, thereby, schedule modules on both higher and lower priority levels. The Set Level Status Block instruction is privileged.

Figure 91. The level status block and module scheduling (2 of 2)

Registers which control the processor and which are not referenced by user tasks are read and written with privileged instructions to prevent inadvertent error on the part of a task.

Registers which are critical include:

- The storage protection register containing the address keys. These keys also select segmentation register sets when hardware address translation is used.
- The segmentation register stacks which map application address spaces into physical storage
- The level status block registers—like the level status register which is not normally set directly by users

Set Address Key Register (SEAKR)
Set Console Data Lights (SECON)
Set Floating Level Block (SEFLB)
Set Instruction Space Key (SEISK)
Set Interrupt Mask Register (SEIMR)
Set Level Status Block (SELB)
Set Operand 1 Key (SEOOK)
Set Operand 2 Key (SEOTK)
Set Segmentation Register (SESR)
Set Storage Key (SESK)

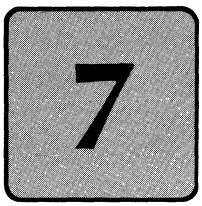
Copy Address Key Register (CPAKR)
Copy Console Data Buffer (CPCON)
Copy Current Level (CPCL)
Copy In-Process Flags (CPIPF)
Copy Interrupt Mask Register (CPIMR)
Copy Instruction Space Key (CPISK)
Copy Floating Level Block (CPFLB)
Copy Level Status Block (CPLB)
Copy Operand 1 Key (CPOOK)
Copy Operand 2 Key (CPOTK)
Copy Processor Status and Reset (CPPSR)
Copy Segmentation Register (CPSR)
Copy Storage Key (CPSK)

Interchange Operand Keys (IOPK)

An application may require a system-like function. Normally, to preserve system integrity, it would be implemented through a Supervisor Call to the operating system. Critical, dedicated tasks—especially those with custom operating systems—may use the privileged instruction set discretely to optimize performance.

Figure 92. The privileged instructions used to read and write Series/1 system-level registers, and control overall processor performance

function efficiently. However, some OEM and dedicated applications which customize the operating system may choose to use these privileged instructions to implement some special critical operation, add input/output device drivers to the system, or perform other hardware operations. Users should be careful to employ the supervisor mode only when necessary and, then, in a manner which does not damage the carefully-designed system integrity.



Interfacing of User Devices

One of the prime application requirements for small computers is the ability to interface special devices to the computer system in an integrated fashion. Many applications require special devices not available from the computer system supplier; examples include:

- Specialized data acquisition devices
- Process control instruments
- Special operator consoles
- Devices selected to be compatible with existing systems
- Devices which are selected because of cost, maintainability, availability, or other reasons

Unless the user can physically attach these devices to the computer system and use them just like devices supported by the computer vendor, the system loses some of its versatility. Users must have available all input/output capabilities including direct program control, cycle steal, and burst mode input/output modes. Furthermore, the built in self-diagnosing and operating system software support must be so designed that users can add the appropriate driver and other routines to make their devices a part of the integrated system.

Importance of the Processor Input/Output Architecture

The heart of the Series/1 system design is the processor input/output channel itself; the channel is more than just a method for transferring data into and out of the processor with appropriate handshaking for synchronization. It supports:

- Direct program control operations
- Cycle steal operations
- Burst mode
- Interrupt servicing
- Initial program load operations

The channel provides especially comprehensive error checking including timeouts, sequence checking, and parity checking. The system reports errors, exceptions, and status in two ways: 1) by recording condition codes in the processor during execution of input/output instructions, and 2) by recording condition codes and interrupt information byte status data in the processor during interrupt acceptance. To maintain the level of system integrity, the user-device interfaces take advantage of all of these features. The channel physically extends along the backplane of the processor or input/output expansion unit. Attachments plug directly into the backplane sockets. The system connects external input/output devices to the attachment cards via additional connectors on the tops of the cards.

Importance of System Software Architecture

System software is so organized that any device connected to the input/output channel can be accessed in the same fundamental way. Hence, the primary requirement for software support of a given device is a basic driver which can interpret the precise data format transmitted to or from a user device. Both Realtime Programming System software and Control Program Support software facilitate the addition of such routines to the system. IBM has designed the

Series/1 input/output system (hardware and software) to permit user attachments to be added to the system in as straightforward a manner as possible. Furthermore, Series/1 system design has stressed the ability of the user to interface devices without sacrificing the important self-checking and diagnostic features of the system. To make available the full capabilities of the system, user devices may be attached in a variety of ways (Figure 93):

- Through specialized use of standard devices (timers are the most notable example)
- Through interfaces compatible with accepted standards (Teletype and CRT terminals are the most common examples)
- Through the use of programmable basic interfaces (digital input and output with synchronization signals under program control)
- Through hardware interfaces which provide a fully compatible subset of the input/output channel itself (customer direct program control adapter)
- Through isolating and non-isolating adapters which permit direct connection to the input/output
- Through the GPIB Adapter which provides an industry-standard instrumentation link
- Through the channel and all its control lines (channel repower and socket adapter)

In this list, the complexity of interfacing increases from top to bottom. The last item (channel socket adapter) requires a detailed design of interface hardware similar to that provided for standard Series/1 devices; it is so intimately connected to the computer system that it is also responsible for preventing interference with the signals present in the rest of the system. Concurrently, every capability of the input/output system becomes available to the user device including cycle steal and burst mode data transfers.

For slower devices—which can be handled by direct program control input/output programming—the user can very easily implement the basic interface (integrated digital

input/output) in hardware, but to do so in software requires more program intervention. In contrast, connecting a device to the customer direct program control interface requires more external hardware but less internal software because the hardware interface uses only a subset of the input/output channel signal. In other words, the appropriate level of interfacing capability is present and that level depends upon the nature of the device or devices to be interfaced to the Series/1 system.

The purpose of this chapter is to discuss each interface approach individually, and to illustrate its capability and use. Detailed discussion of voltage levels, loading restrictions, and similar considerations important in the actual design of interface hardware is available in the appropriate Series/1 processor User's Attachment Manual.

Timers and Their Use

Many applications involve the measurement of time intervals, or the counting of events which occur in a given time interval. Manufacturing control applications involving piece count, monitoring of machine operations, and control of material handling systems are common examples. The IBM Series/1 timer feature is a single, printed-circuit card which plugs directly into the backpanel of the processor or input/output expansion unit. Each card contains two timers and as many cards as desired may be used in a system.

Connectors on the card—in conjunction with the required programming—allow the timers to be used with external control signals; the card can then be used as an interval timer (with internal or external clock), a pulse counter, or a pulse duration counter depending upon the configuration of the external signals. Each timer is separately addressable as an input/output device and—without stopping the timer—can be started, read, or set to any value, independently, under program control. These characteristics result in a flexible interface device which solves many application problems with a minimum of special hardware and a maximum of software flexibility.

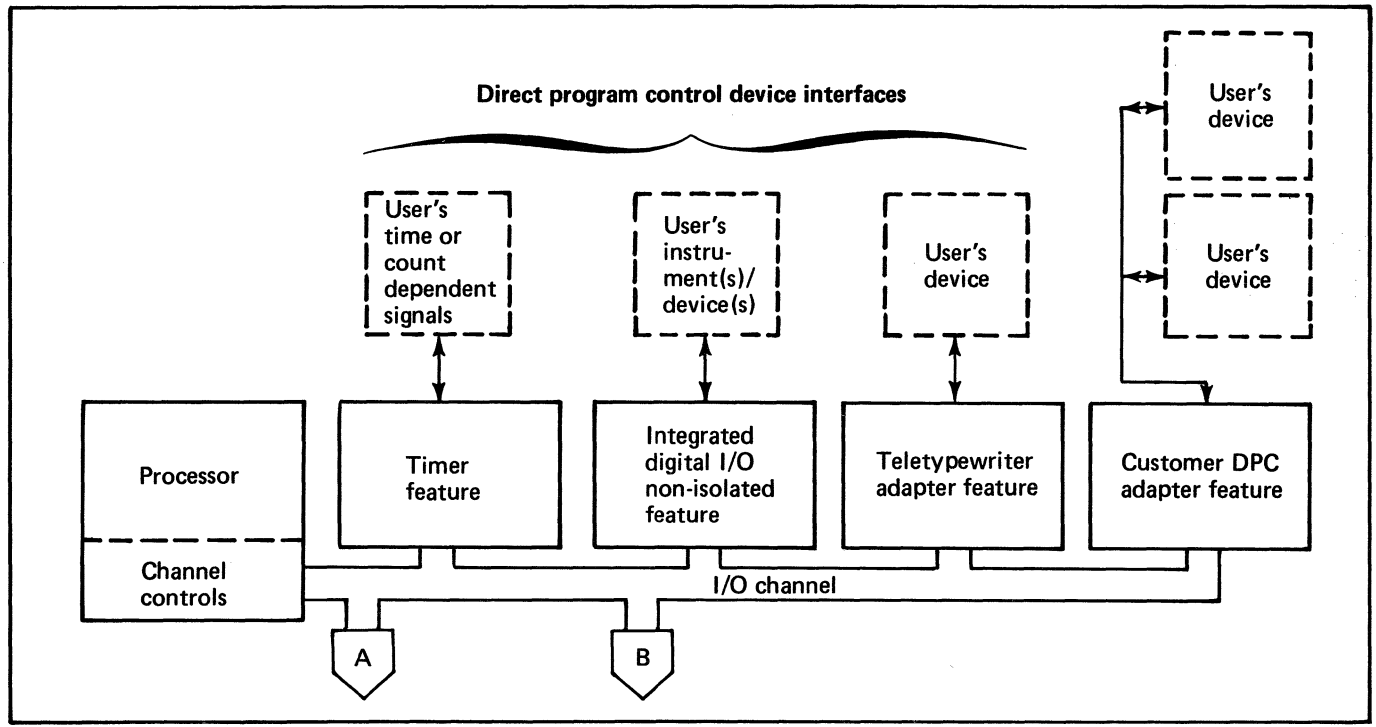


Figure 93. Options for user attachments to the Series/1 (1 of 2)

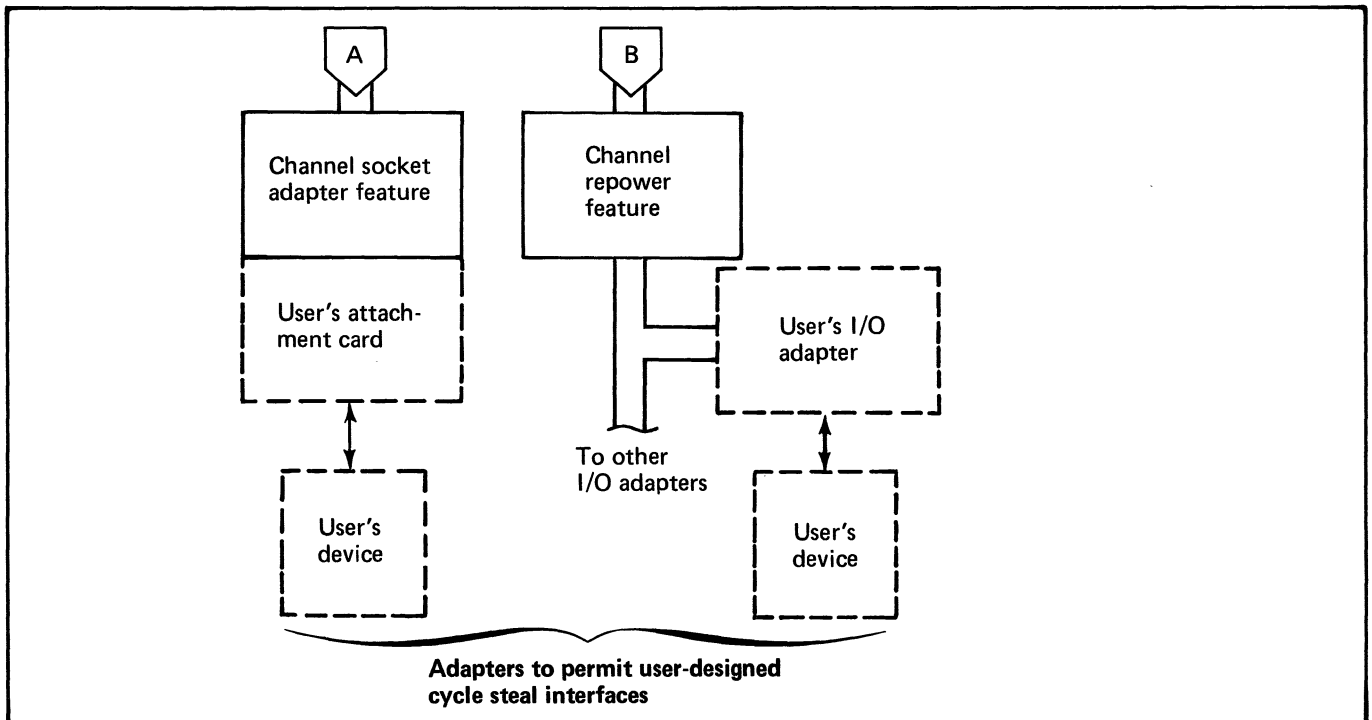


Figure 93. Options for user attachments to the Series/1 (2 of 2)

Each timer has a mode register which the system uses to select the internal time base or to specify an external base. Available internal time bases or increments between counts are: 1, 5, 25, and 50 microseconds. The system selects a time base which provides adequate precision in the desired time measurement. The registers are 16-bits wide so that a maximum count of 65,535 bytes is available. For the internal time bases listed above, this maximum count corresponds to time intervals of approximately 65 milliseconds, 328 milliseconds, 1.6 seconds, and 3.3 seconds. Four program selectable running modes are available for each timer:

Periodic Interrupts—Internal Control. Program control sets a 16-bit autoload register to any value. This register automatically reloads the timer when the timer underflows, and the system generates an interrupt.

Aperiodic Interrupts—Internal Control. The system loads the timer with a value under program control, and an interrupt occurs when the timer underflows. After the first interrupt, the autoload register does *not* reload the timer.

Periodic or Aperiodic Interrupts—External Control. The timer generates periodic or aperiodic interrupts, but—when the timer is in the run state—an external gate signal controls timer start and stop.

Figure 94 shows a block diagram of the timers and their connections to the input/output bus. Notice that the address portion of the bus is used to address the timers and to transmit specific commands like setting mode and counts. Address jumpers permit assignment of an arbitrary address to the pair of timers. As depicted in the figure, the system uses the data bus to communicate with the mode registers. The interface to external signals of the user is through drivers and receivers which provide isolation and adequate current for compatibility with commonly used external electronic circuitry.

The external process devices may supply two signals: a clock signal and a gate signal. A clock is a source of pulses or event signals which may be periodic or aperiodic depending upon the application. Periodic clock signals provide time

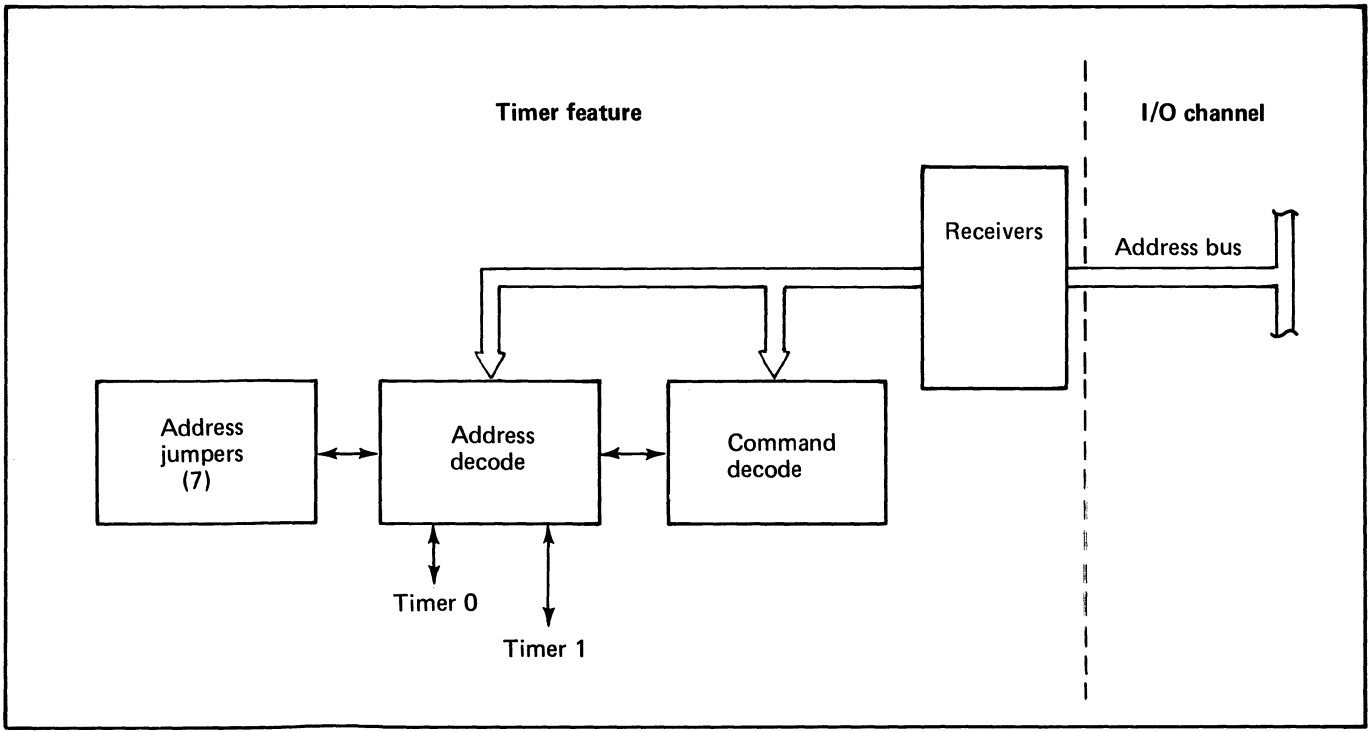


Figure 94. Block diagram of the timers showing their input/output channel connections and external signals for special uses (1 of 3)

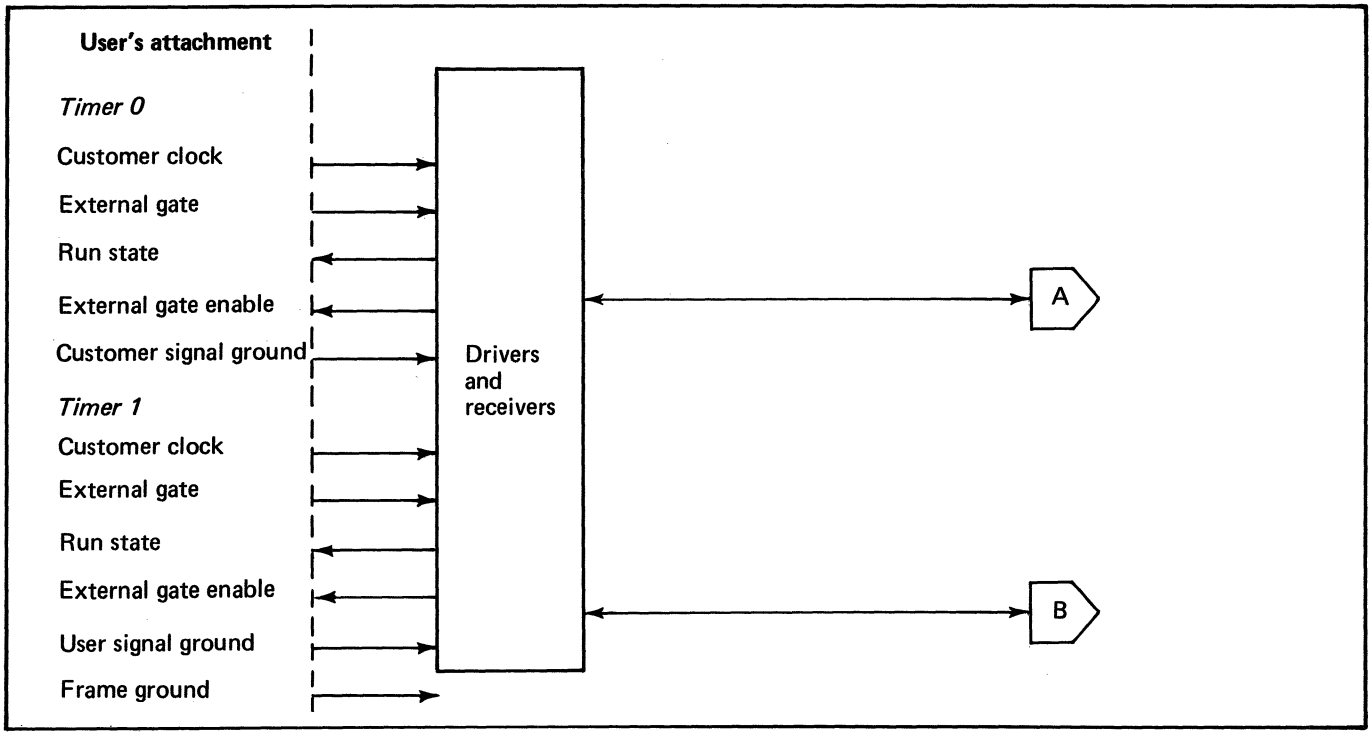


Figure 94. Block diagram of the timers showing their input/output channel connections and external signals for special uses (2 of 3)

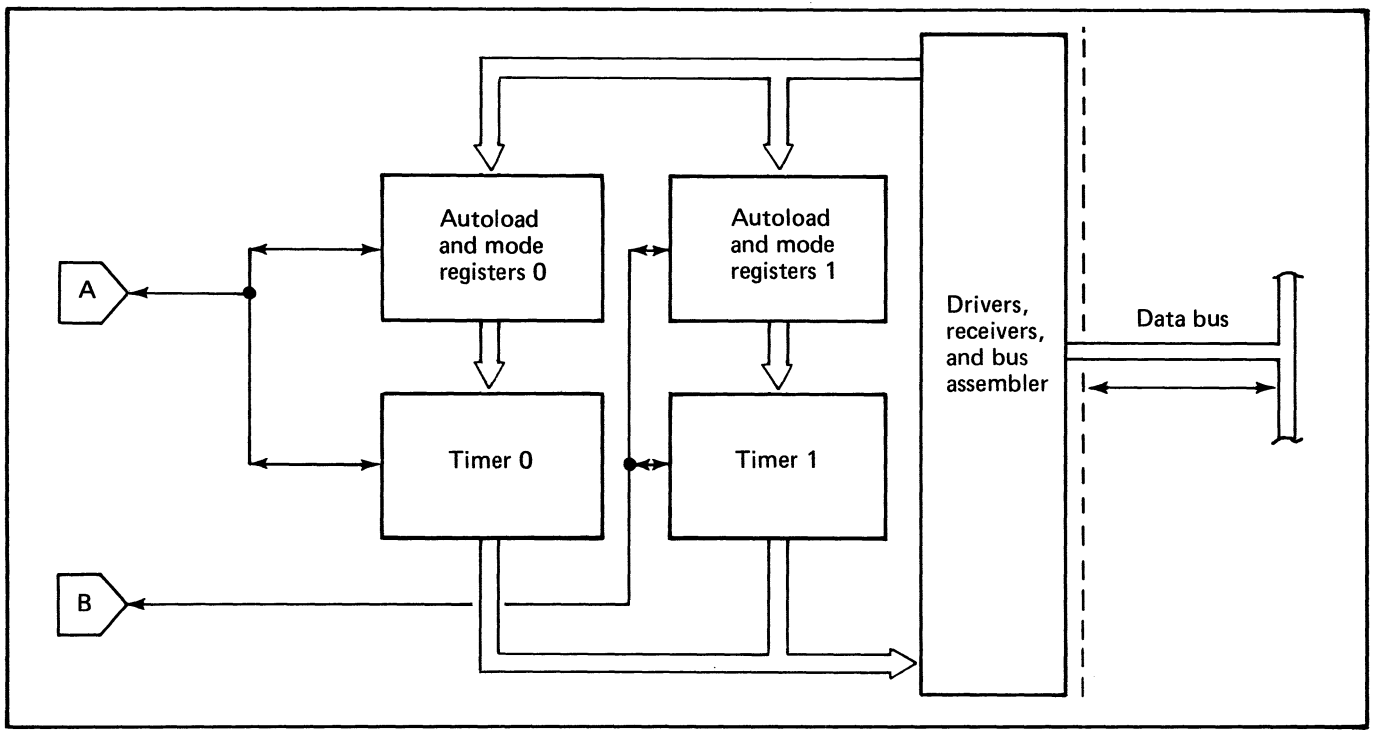


Figure 94. Block diagram of the timers showing their input/output channel connections and external signals for special uses (3 of 3)

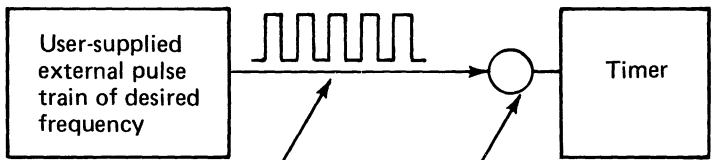
bases which are different from internal bases. Applications involving counting of events where one pulse is generated for each event that occurs use aperiodic sources. The gate is a signal which indicates when to begin and when to stop counting. The two output control signals, run and external gate enable, indicate to the external device that the timer is active and that the external gate has been enabled for use. These signals permit synchronization between the internal application task using the timer and the external device supplying the signals.

Interval Timing

Figure 95 shows how the system uses the timer to provide interval timing to the processor by using an external, arbitrary time base. The external clock is attached. Under program control, the system prepares the timer (sets priority level and enables device interrupts). Its mode is set to external-periodic. This means that the clock pulses come from the external source; every time the timer counts through zero, an interrupt is generated to indicate the end of an interval, and the counter is reset to its original value. Finally, the system transmits a command (direct program control input/output operation) to start the timer. Using the Realtime Programming System, the Event Driven Executive, or Control Program Support software, the user can attach a task to this interrupt which then becomes active each time the pre-set interval expires. The external clock can be as slow as desired and can be as fast as 20 microseconds (the internal clock permits the maximum precision of one microsecond). The timing interval can be changed under program control.

Pulse Rate Measurement

Another common application is shown in Figure 96 where the user needs to measure a pulse rate. This is usually done by counting the number of pulses which actually occur in a given time interval. As illustrated in the figure, the system counts the pulse source in one of the two timers on a card. This timer is loaded with a value larger than the maximum



Customer clock input signal line (grounds not shown)

Clock frequency can be as arbitrarily slow or as fast as 50,000 pulses per second. Faster clocks up to 1,000,000 pulses per second are available internally in the timer if the filter is inactive.

Figure 95. Using the timer to provide interval timing to the processor (1 of 3)

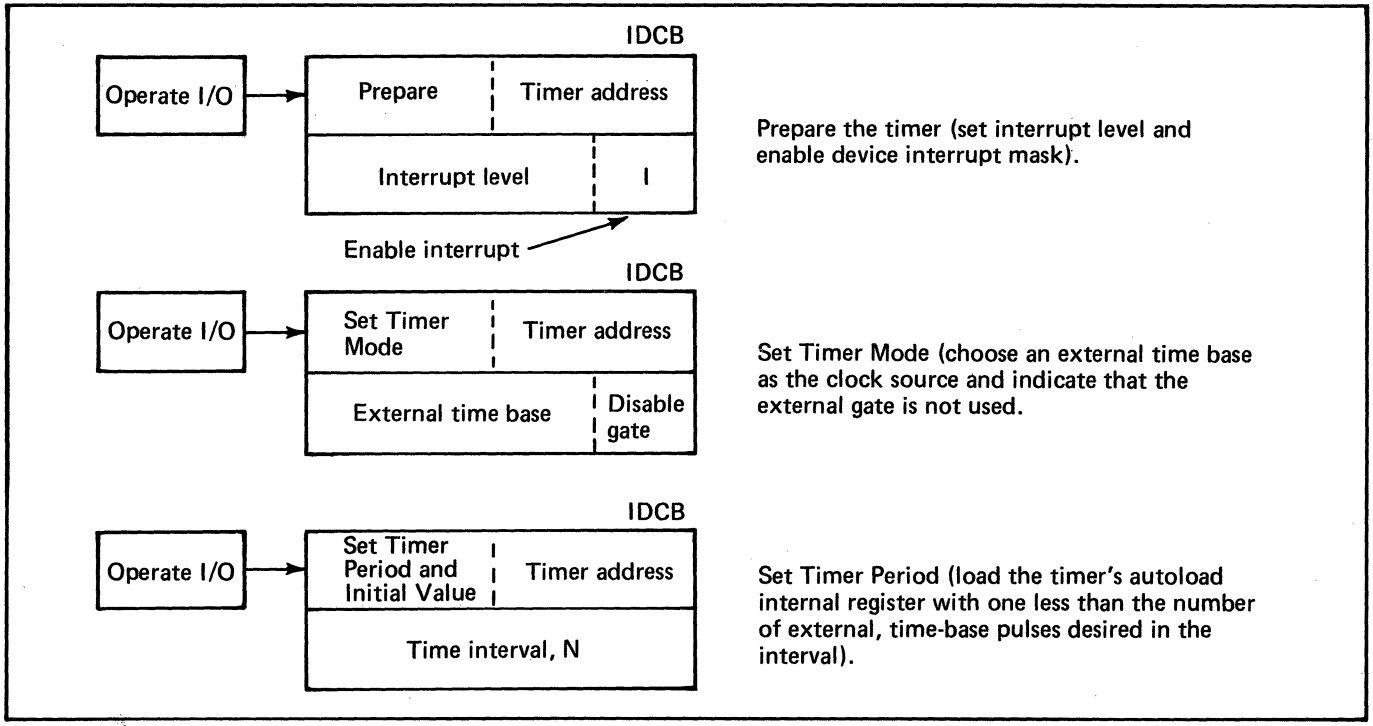


Figure 95. Using the timer to provide interval timing to the processor (2 of 3)

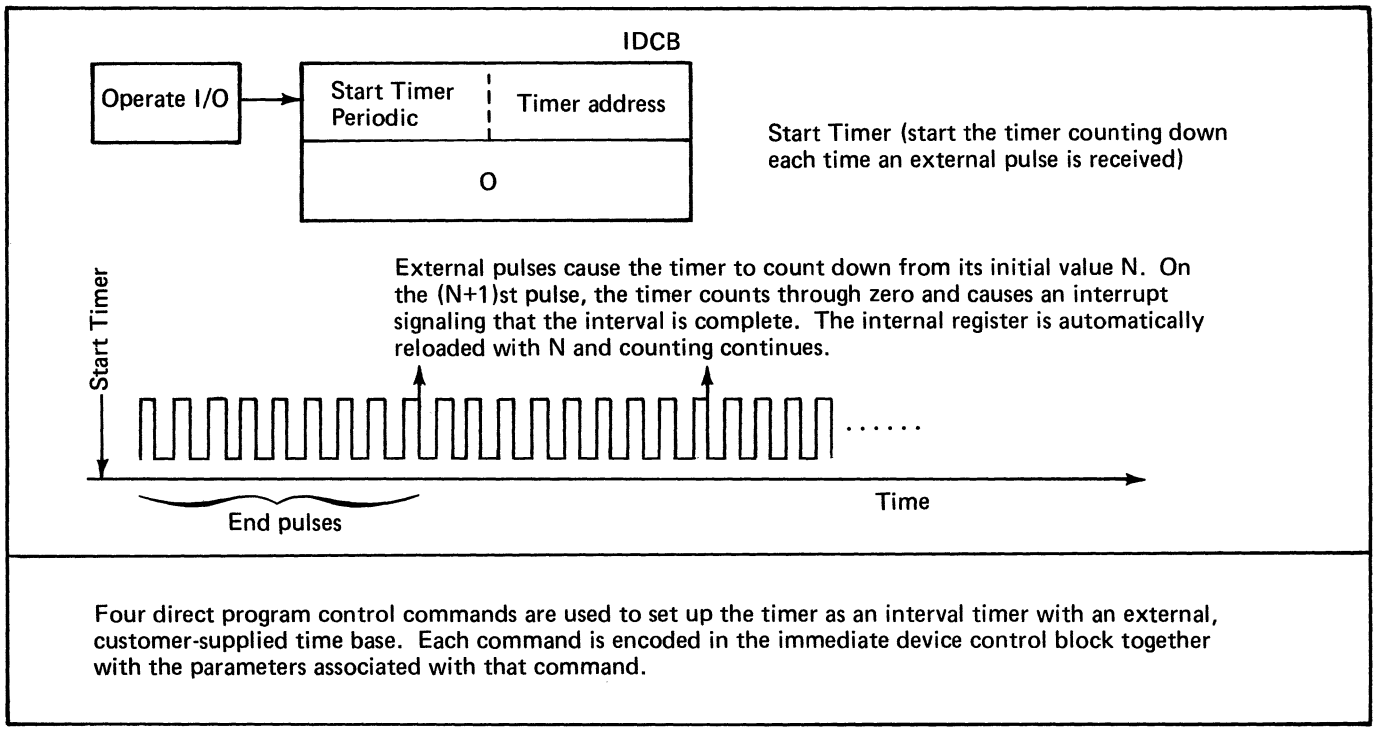


Figure 95. Using the timer to provide interval timing to the processor (3 of 3)

Problem: Measure a pulse rate in an external sequence of pulses which may be irregular in separation.

Solution: Measure the pulse rate as the number of pulses per unit time. Use a timer as an interval timer to set the basic time period, and a second timer to count the pulses occurring in that period.

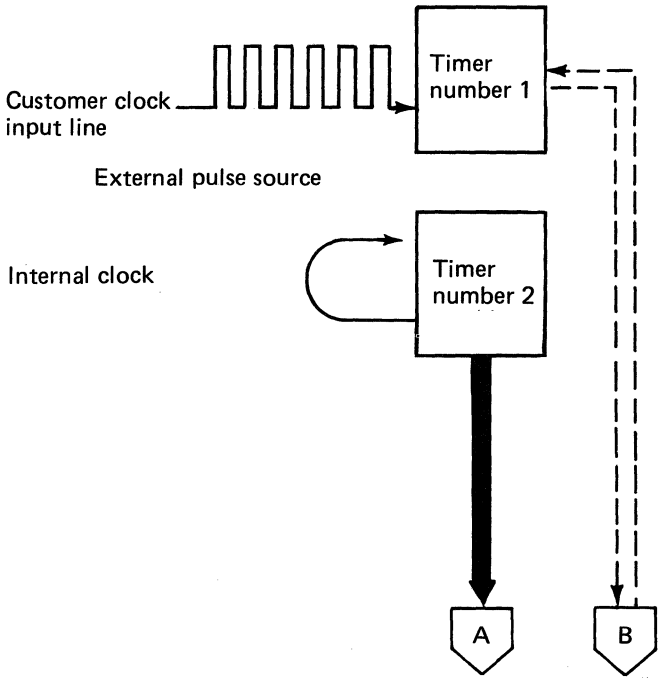
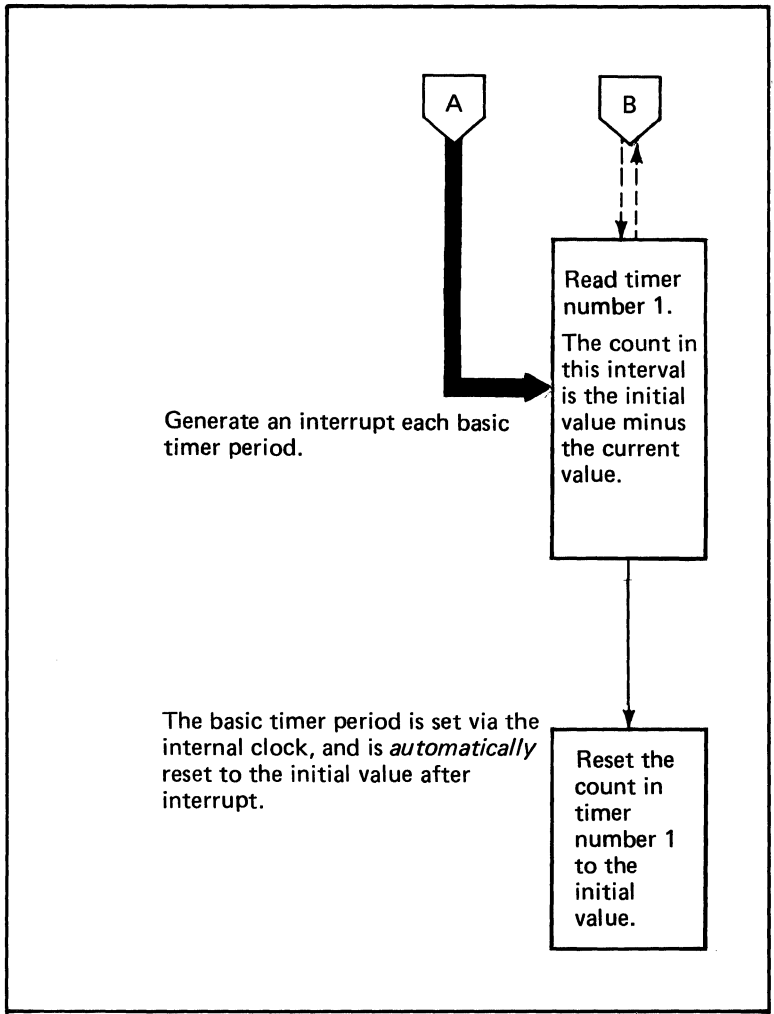


Figure 96. Pulse rate measurement using a pair of timers (1 of 2)

number of counts expected in the measurement time interval; as each pulse is received, the timer counts down but does not pass through zero or interrupt. When read, this timer contains the original value less the number of counts received



Initialize:

- Timer number 1 to a value larger than the maximum expected pulse-count-in
- Set mode to the external time base
- Timer number 2 to the internal time base, periodic mode; and initialize the count to the correct basic period

Figure 96. Pulse rate measurement using a pair of timers (2 of 2)

during the interval. Hence, subtracting the value read from the initialized value gives the counts received.

As depicted, a second timer generates the interval used in the rate measurement. The second timer is set up to operate in the internal-periodic mode which: 1) generates an interrupt each time the base measurement period passes, and 2) initiates a task that reads the first timer and calculates the rate. By choosing the required internal clocks, two timers are adequate to meet the needs of most applications of this type.

Pulse Duration Measurement

Pulse duration measurement is equally straightforward using the external gating capability of the timer (Figure 97). The system sets up the timer to operate under external gate control. In this mode, an interrupt occurs when the external gate is turned off—at the end of the pulse being measured. Meanwhile, the system sets up the timer counter with a value larger than the number of counts expected during the pulse. The counter actually starts when the external gate signal is turned on; counts as long as the gate signal is present; and stops counting when the gate signal is turned off. Responding to the interrupt and reading the counter value give the number of pulses. Knowing the period or time between pulses then yields—with a precision corresponding to the time base used—the pulse duration.

Error Detection

It is important to be able to detect abnormal timer operation in applications like these; otherwise, a critical realtime application may not function properly under certain circumstances. Normally, the timer has decremented through zero (counted one more than the initial count set in the register), or the external gate signal has been turned off. However, an error condition can occur under several situations, including:

Overrun. The interrupt has occurred while the previous interrupt is still pending; this means that the processor has not

yet responded to the first interrupt. This situation causes loss of data or a mistake in time tracking.

Counter Zeros with External Gate Enabled. The counter decrements through zero—generating an interrupt—while the external gate signal is enabled and not yet turned off. In the pulse duration application, for example, the count will not be the accurate pulse length. The initial value set in the counter was too small or an unexpectedly long pulse occurred. In either case, the situation must be specially handled.

As part of the self-checking capability of the timer and the Series/1, the system reports each condition via a different condition code. Consequently, the task responding to the interrupt can:

- Check the condition code
- Determine whether an error or special case is present
- If it is, handle it appropriately

The Teletypewriter Interface

The teletypewriter adapter is an interface designed to attach OEM devices which operate as start-stop devices in full-duplex mode over a four-wire interface. Data can be transferred in current loop mode, TTL[1] standard signal mode, or EIA[2] standard mode. Typical devices designed to interface by one or more of these procedures include:

- Printer keyboards
- Keyboard displays
- Keyboard-display printers
- Printers
- Tape cassettes
- Tape drives

[1] A common electronic technology
[2] Electronics Industry Association

Problem: Measure the length of an external pulse (the time it is on). Alternatively, it may be necessary to measure—very precisely—the time between two external events.

Solution: If the time interval is long enough, an interrupt can be generated at the beginning and at the end; software can be used to read the internal system clock. If the events are close together, however, this technique has limited precision.

For greater precision, generate an external pulse which is:

- Off before the start of the pulse or first event
- Turned on at the start of the pulse or first event
- Turned off again at the end of the pulse or second event

Use this signal as an external gating signal to turn the timer counter on and off.

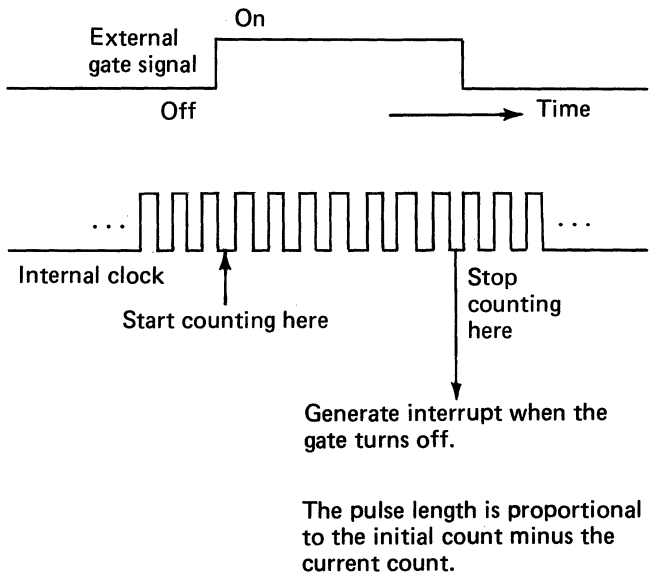


Figure 97. Pulse duration measurement using the external signal and a timer (1 of 2)

Initialization:

- Prepare the timer interrupt level and enable interrupts
- Set the mode to internal clock and enable the external gate
- Set the initial value into a register which is larger than the maximum number of pulses expected during the duration of the pulse

If the pulse duration is too long, the timer will count through zero and generate an interrupt. Condition code specifies that the gate was still enabled and that it must be checked to detect errors.

Figure 97. Pulse duration measurement using the external signal and a timer (2 of 2)

- Card readers
- Badge readers
- Plotters

Every application mentioned in Chapter 1 uses devices like these; they are available from a very large number of manufacturers who specialize in various applications. The teletypewriter interface solves many application problems and is, consequently, an important hardware entity.

Asynchronous Data Transmission

The term start-stop data transmission means that one character at a time is transmitted, bit serially, in either direction. Figure 98 shows the format of the transmission which includes a start bit located prior to the eight bits of the character, and either one or two stop bits following the character. Between characters, the system holds the line in one logical condition called "mark" and signals the start of a character by the transition to the other logical condition called "space", which is also the start bit. Spacing between characters is arbitrary, but bits within a character are synchronous. Bit rates of 50, 75, 100, 110, 150, 200, 300, 600, 1200, 2400, 4800, and 9600 bits per second are standard and selectable—by jumper pins—on the interface card.

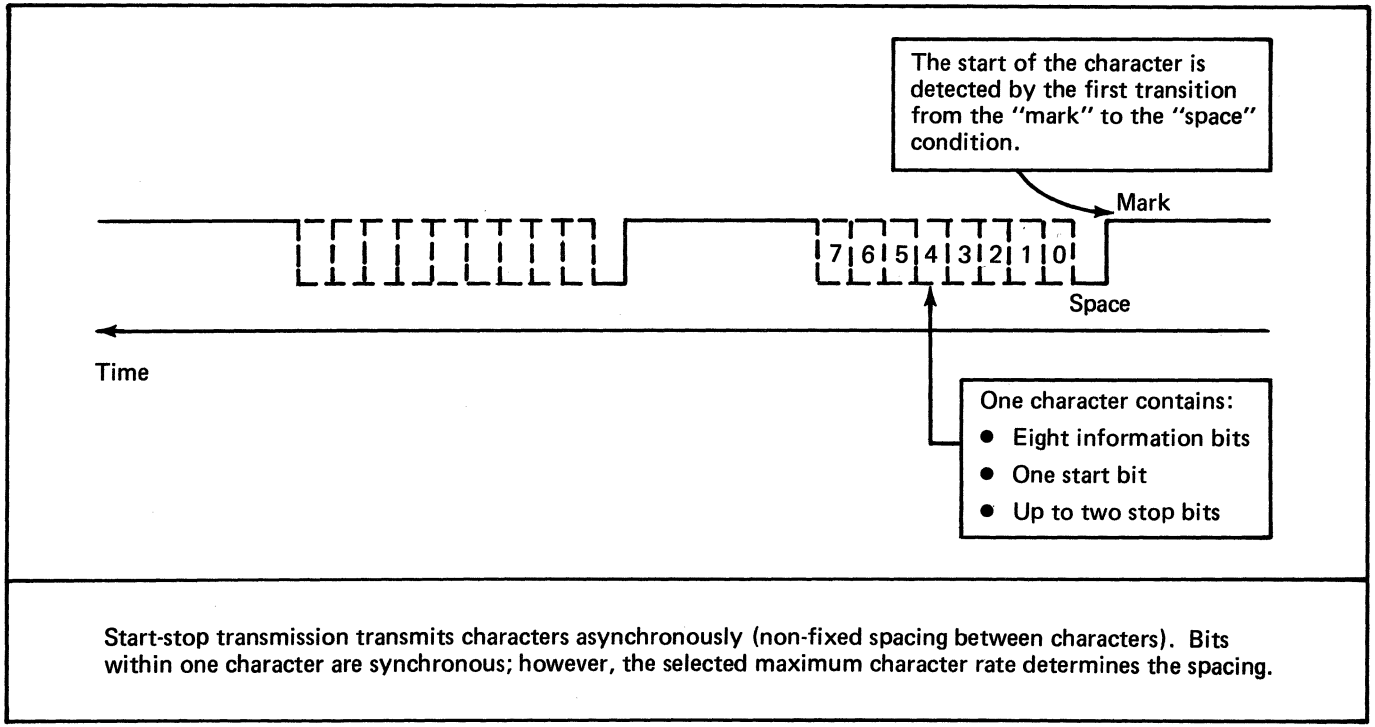


Figure 98. Start-stop character transmission

The system performs no error checking on the byte being transmitted. The bits of the character are received one at a time and assembled into a character. The system generates an interrupt and transfers the character into the processor by one direct program control operation. Since no error checking is done, all 256 possible character combinations are legal characters and the support of this device is "code transparent;" that is, it is not dependent upon the meaning of the characters. This fact is important when using devices like those listed at the beginning of this section because many manufacturers assign special meanings to certain characters—especially in CRT devices. The user task can read and transmit arbitrary character sequences from and to these devices, but it is also responsible for interpreting the meaning of the characters.

Each time a source receives or transmits a character, the system generates an interrupt. An overrun can occur on reception of characters but not on transmission. Overrun, in this case, means that a source receives a second character before it recognizes the previous interrupt. The processor has not yet read the previous character. The interface detects this condition and signals the processor with an exception interrupt (via the condition code presented in the level status register). When overrun occurs, the first character is not lost but the second is.

The Asynchronous Interface

Figure 99 shows a block diagram of the interface and its connection to the input/output channel of the Series/1 processor. Notice that normal reading and writing of characters occur only under direct program control, but that the system can perform initial program load on a cycle steal basis. The latter technique is useful for devices like tape cassettes which can store initial program loads in a safe manner and transmit them quickly during startup. The initial program load mode is standard in the teletypewriter interface.

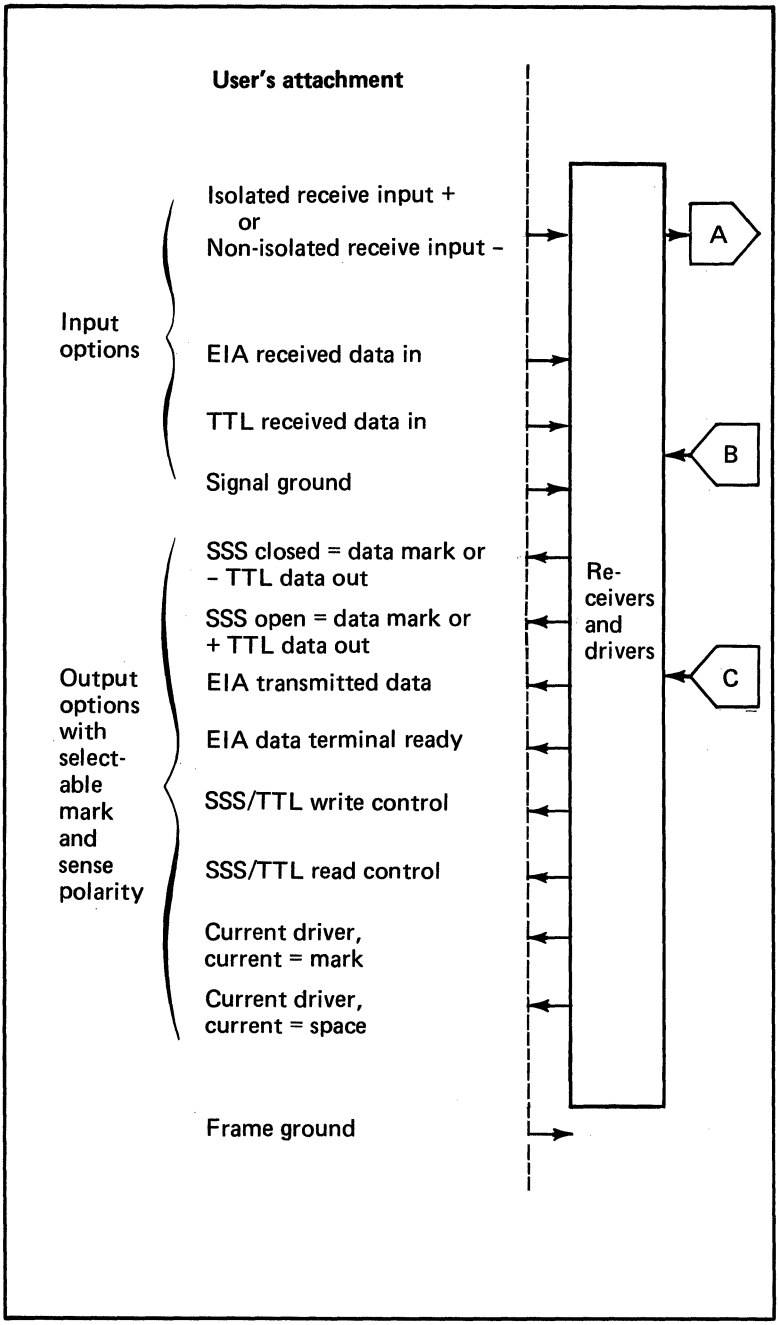


Figure 99. The teletypewriter interface block diagram (1 of 4)

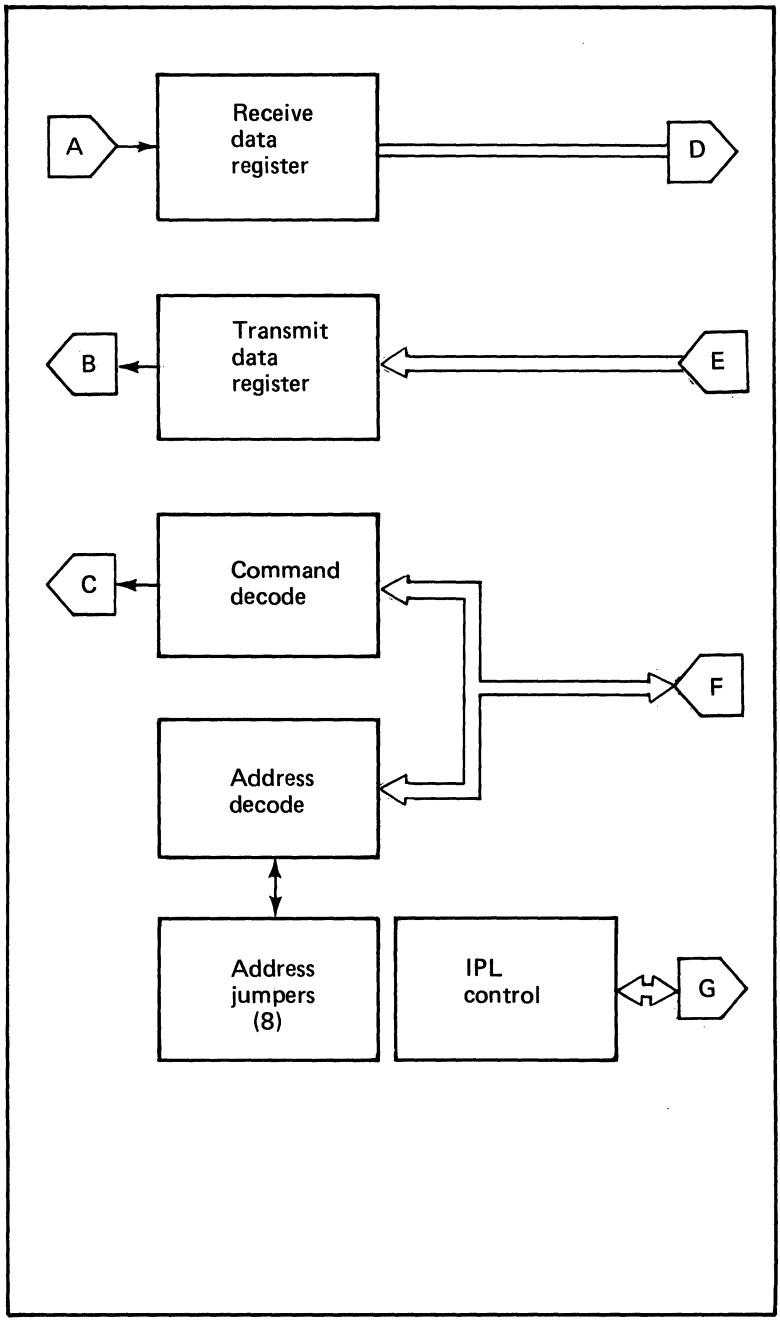


Figure 99. The teletypewriter interface block diagram (2 of 4)

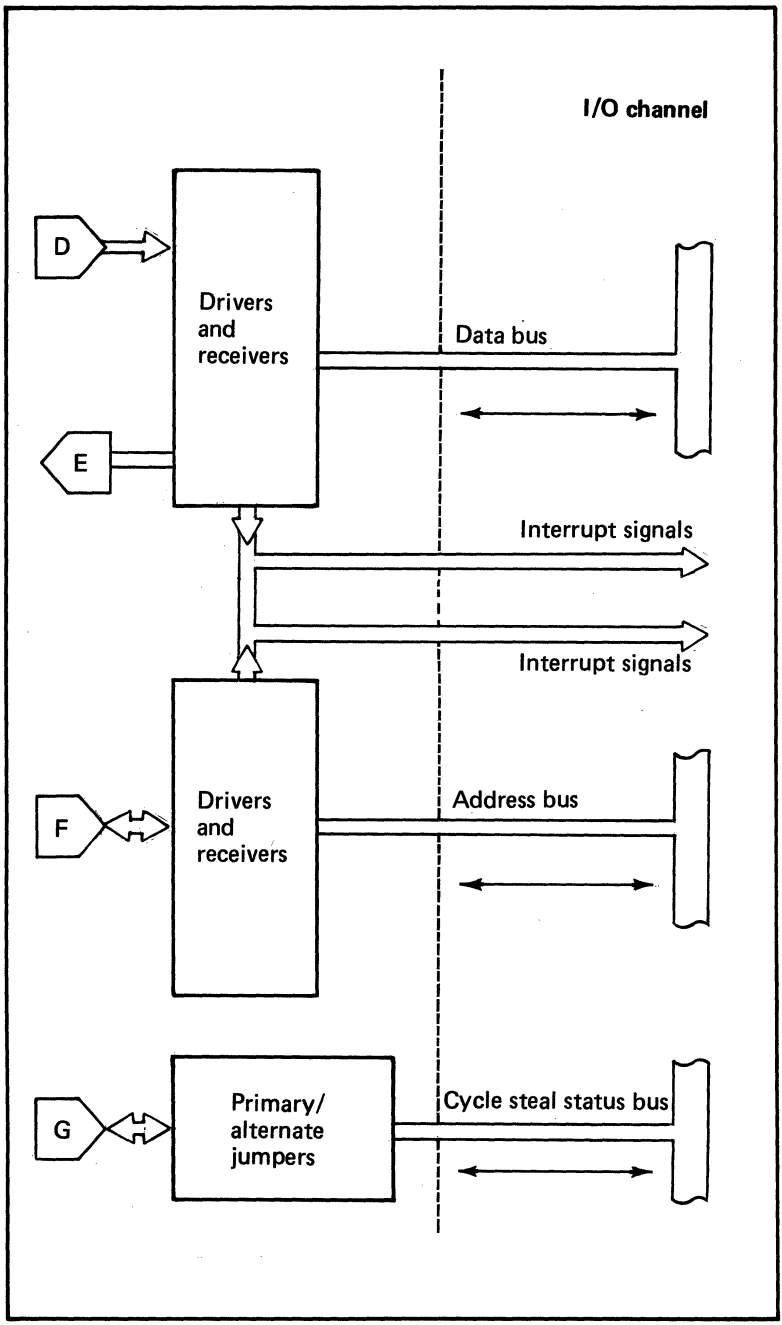


Figure 99. The teletypewriter interface block diagram (3 of 4)

The interface does not interrupt any characters. Each character input causes an interrupt. The processor responds to the interrupt and reads in the character.

All character transfers are one character at a time under direct program control. Initial program load is supported by the interface on a cycle steal basis.

Abbreviations:
TTL = transistor-transistor logic
EIA = Electronic Industries Association
SSS = solid state switch
IPL = initial program load

Figure 99. The teletypewriter interface block diagram (4 of 4)

As in all Series/1 interfaces, jumpers on the interface select the device addresses; position on the processor interface or input/output expansion chassis is not relevant to the selection. The interface signals to external devices are shown under the heading "User's attachment" in Figure 99. These connections are made physically through a 16-pin connector on the interface board. Jumpers on the interface itself select the mode of interconnection along with the bit rate and device address.

To be compatible with the variety of devices available, Series/1 supports several input and output modes as shown in the figure:

- Inputs
  - Non-isolated contact sense
  - Isolated contact sense
  - TTL signal levels
  - EIA signal levels
- Outputs
  - Current driver
  - Solid-state switch with TTL signal levels
  - EIA standard signal levels

In addition, the interface permits the system to select the mark condition at either polarity in the case of voltage level inputs, and at either open circuit or short circuit in the case

of a non-isolated contact sense. Similarly, outputs which use the current mode may select either a presence of current condition or a lack of current condition to indicate the mark. Solid-state switch or TTL outputs permit either polarity or open or short circuit conditions to represent a mark. EIA output uses the standard convention of negative voltage implying a mark condition. The many different options available also eliminate annoying, small hardware "fixes" that some systems require to make a terminal or device compatible with so-called standard interfaces.

If it is needed to solve special problems, the synchronization signals (solid state switch and TTL write and read control signals) allow program control of external devices.

Software Support

The reader should note again that the code transparency of this interface means that—in order to give full software support to these devices—user-written tasks must be supplied and integrated with either the Control Program Support routines, the Event Driven Executive, or the Realtime Programming System. Notice further that no error or diagnostic information is present on the external interface signal lines. The interface itself, of course, is IBM-supplied and contains the usual self-diagnostic features mentioned previously. Diagnosis of the device itself—once the interface has checked itself—is the responsibility of the device and the user-written software. For devices which have been designed to accept diagnostic commands, the code transparency permits the user to achieve the same high level of device checking for OEM devices as is performed in IBM-supplied devices.

The Integrated Digital Input/Output Interface

The timer and teletypewriter interfaces discussed in the sections "Timers and Their Use" and "The Teletypewriter Interface" can handle most of the data transfer, synchronization, and error detection for the devices they support because they are designed for very specific device classes. At the

circuit level, the system automatically processes sequences of events—like data presentation on signal lines and handshake signals—because the system knows the event sequence for the class of devices supported. As interfaces become more general, event sequences become more varied and uncertain; and users must handle more and more of the procedures by themselves. Basically, all interfaces involve:

- Transmission and reception of data on signal lines
- Timing of handshake signals until data has settled, or until a signal has been received
- Similar operations which entail
  - Inputting and outputting of binary or digital data
  - Setting and resetting of control lines
  - Generating interrupts to signal event occurrences

The integrated, digital input/output interface provides the basic interfacing capabilities: groups of digital input and output lines, together with control lines which the system sets or resets under program control. With this interface, the system can attach any device if the sequencing of actions is performed in software using direct program control input/output instructions. This interface is similar to those previously described except that the user must program the detailed operation to conform with the particular device attached.

Structure of the Digital Input/Output Interface

Figure 100 shows the integrated digital input/output interface in block diagram form. The interface includes two 16-point groups of non-isolated digital input coupled with process interrupt, and two 16 point groups of non-isolated digital output. The four groups each have a unique device address but are prepared as a group (that is, they are enabled or disabled as a group and have a common interrupt level). External synchronization signals (external sync and ready lines in Figure 100) are available for each group. A single card contains the interface which is pluggable into any input/output slot. Up to four such interfaces per card file may be used.

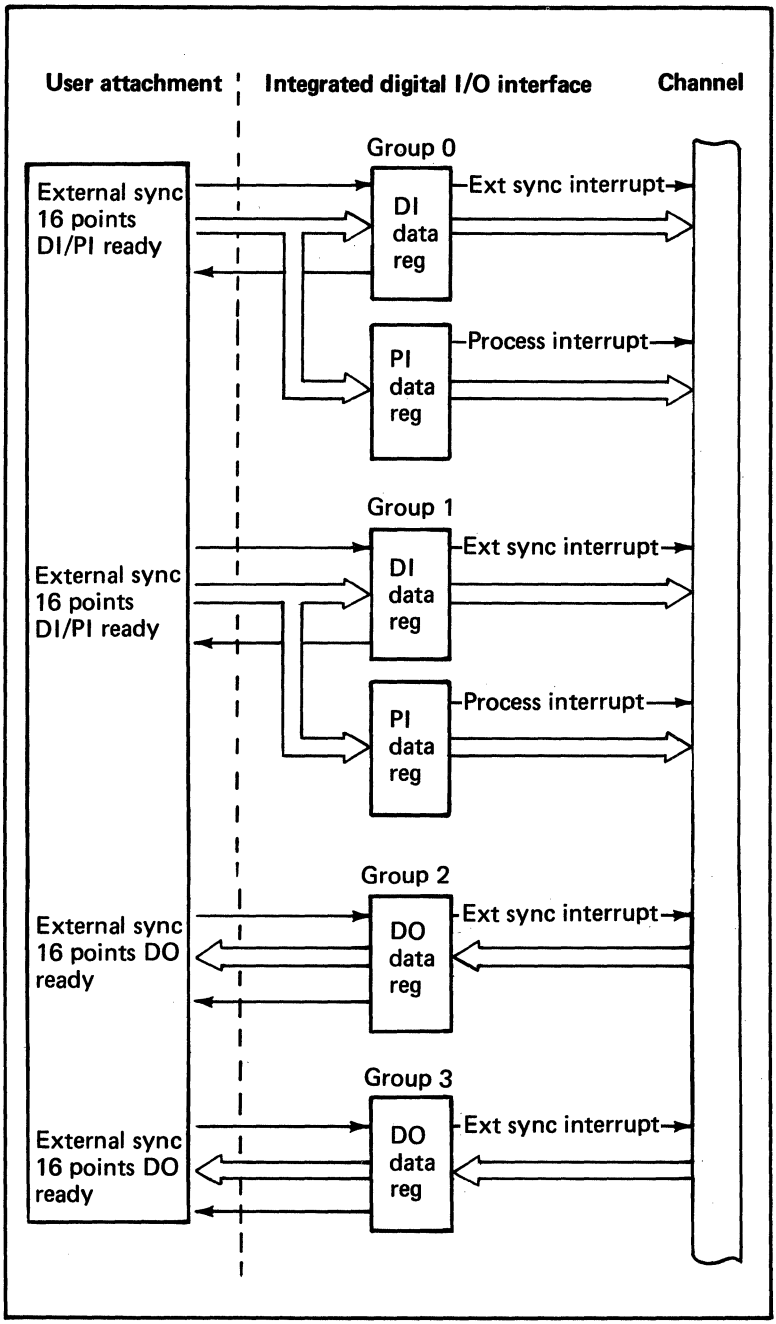


Figure 100. Integrated digital input and output interface (1 of 2)

Each of the four 16-bit groups has its own device address.

The four groups operate on a single interrupt level, and have a single device-interrupt control bit. All four groups are prepared with a single command.

Figure 100. Integrated digital input and output interface (2 of 2)

Digital Output

Consider first the digital output groups. Each 16-point group provides:

- Non-isolated unipolar current switches or TTL voltage switches
- The two control signals
- An interrupt capability from the external sync output line

The system stores data in the digital output register shown in Figure 100 using a direct program control Write command. Digital output operates in three modes: 1) non-interrupting, 2) external sync, and 3) diagnostic. In the non-interrupting mode, any data present in the digital output register is presented to the output lines and consequently to the connected output device. This mode is useful to signal display registers or operate solenoids or electrically operated switches, and to signal those lines which do not require handshaking between transmission and reception.

External Device Synchronization

In the external sync mode, set by an Arm direct program control input/output command, the system sets up a handshaking communication using the external sync and ready lines associated with each group (Figure 101). Under program control, the system prepares the digital output group (interrupt level and interrupt enabling set), arms the external sync mode, and outputs data to the register. The ready line is held reset until the external sync line sets, signaling that a transfer may take place (this line is set by the external device). At this time—after the data written to the output

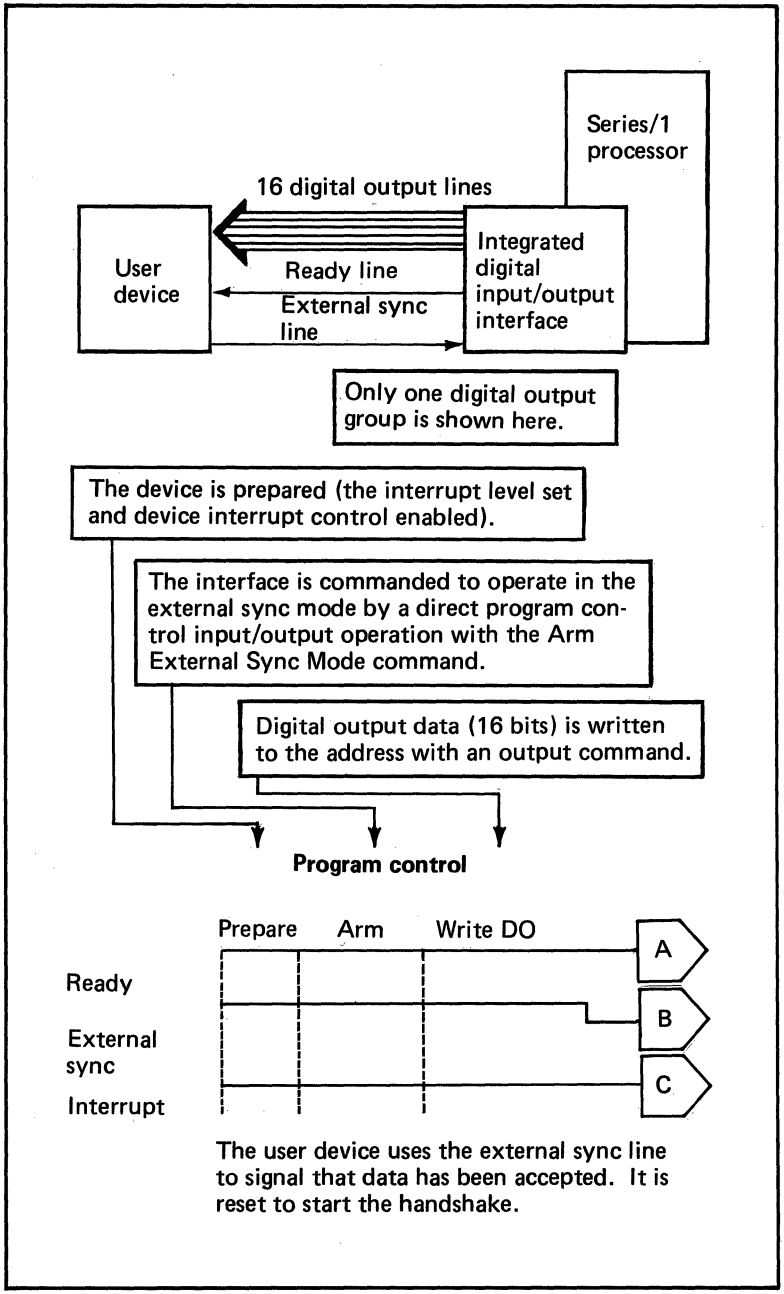


Figure 101. The handshake convention used on digital group output (part 1) (1 of 2)

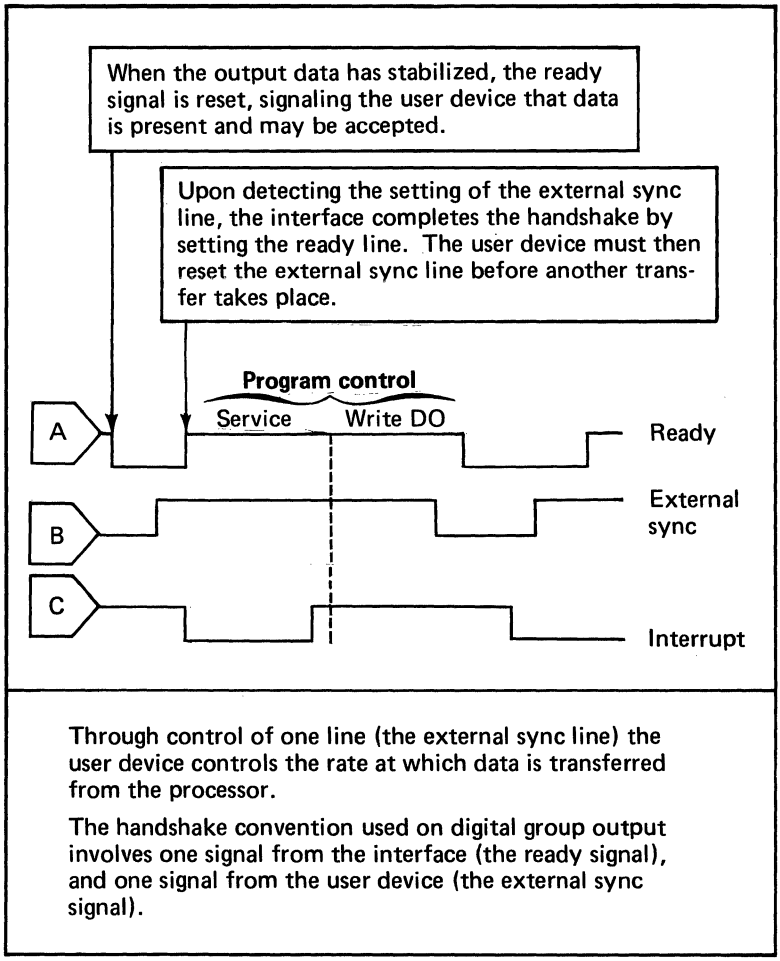


Figure 101. The handshake convention used on digital group output (part 1) (2 of 2)

register has settled—the ready line is set, signaling the external device that the data is ready to be read. When the external device has accepted the data, it resets the external sync line, signaling the interface that it has received the data. The interface then resets the ready line and the system can initiate another transfer. The system accomplishes synchronization with the task, writing the data by generating an interrupt when the resetting of the external sync line is detected; this interrupt indicates that the data has been successfully transferred.

Digital Input

The digital input groups are more complex in their operation primarily because they have the ability to latch or remember changes at their inputs, and to cause interrupts. Each of the two digital input groups has a unique device address. The digital input registers in each group track the levels of the external points attached to them. This tracking does not occur during a read operation or in external sync mode when the values are held constant within these parameters: after an interrupt is generated, and until the interrupt is accepted. The process interrupt registers—also associated with each input group—latch or remember any bit transition from zero to one; such a transition generates an interrupt.

Non-Interrupting Mode

The digital input groups operate in one of four modes: 1) non-interrupting, 2) process interrupt, 3) external sync, and 4) diagnostic. In the non-interrupting mode, Read commands can reference either the digital input or process interrupt registers in the group, and simply read their current contents (which cannot change during the read operation). Hence, reading the digital input gives the current state of the sixteen input lines; and the Read Process Interrupt command gives the current state of those lines which have experienced a zero to one transition at any time since the system reset the register.

Process Interrupt Mode

The process interrupt mode is entered by a direct program control input/output operation—Arm process interrupt. Any zero to one transition sets the corresponding bit in the digital input group process interrupt register and generates an interrupt. Accepting the interrupt, and performing a Read Process Interrupt Register input/output command, reads the register contents and then resets it. By performing

a Shift and Count instruction, the system can determine—with a single instruction—the particular bit which changed. This mode is very useful for implementing multiple interrupts from a device or devices. In such an application, reading the digital input group could be used to test the status of the input lines. That is, if more than one line changed, the system would generate an interrupt and read the process interrupt register. This action, however, resets the whole register; whereas the digital input register tracks only the input lines that are still set at one.

External Sync Mode

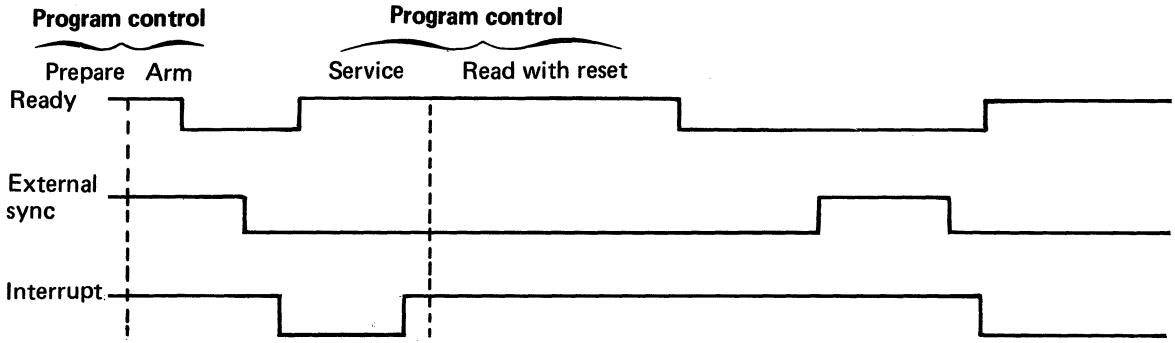
External sync mode provides the same kind of handshake communication for digital input as it does for digital output (Figure 102). This mode is entered with a special input/output command to the interface. When in this mode, the system activates the ready control line, indicating that the system is ready to receive input data. The external device detects the ready signal, puts data on the sixteen input lines, and sets the external sync line. This transition:

- Causes the system to latch the data or hold it constant
- Reactivates the ready line
- Causes an interrupt to the processor

The application task or driver responds to the interrupt, and reads the latched digital input group with a normal Read Direct Program Control command; this action, in turn, causes the system to reactivate the ready line and unlatch the input register so the latter is ready to receive more input data. Each end of the communications' line signals the other with these two control indicators.

Control over a device, then, usually involves a series of program-controlled transfers in and out of the processor. The system monitors device control signals so it knows when:

- Data is ready or data can be accepted
- Data is transferred in the appropriate direction
- Status checking is performed



Digital input to the processor from a user device is handshaked with two signals in a manner similar to digital output. Digital input is interrupt driven.

Figure 102. The handshake convention used on digital group output (part 2) (1 of 2)

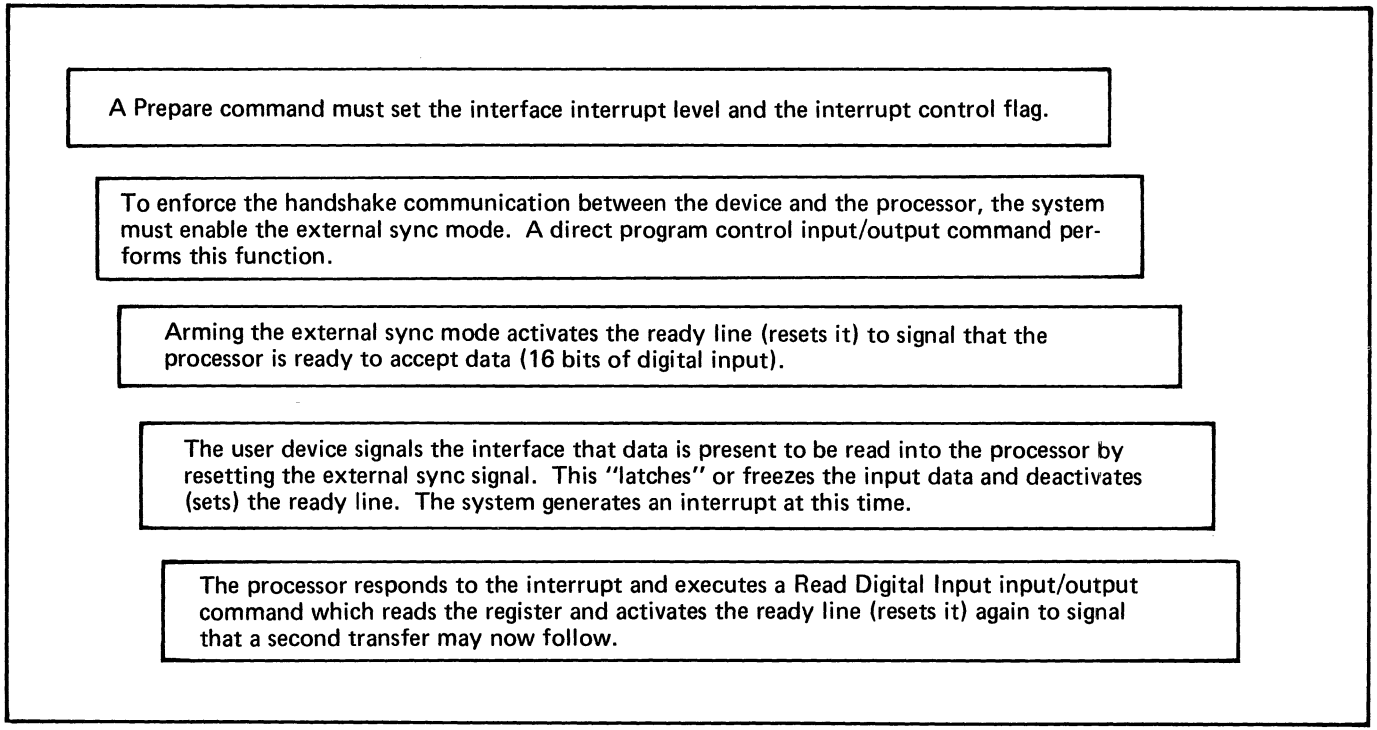


Figure 102. The handshake convention used on digital group output (part 2) (2 of 2)

Instead of the interface performing this function in parallel with the processor operation, each step of the operation may involve input/output instructions. Although its efficiency is lower, the flexibility of the component allows a user to interface complex, slower devices with a minimum of both hardware and software. Such a technique is often economical and always versatile.

Diagnostic Mode

The integrated digital input/output interface provides a full set of diagnostics. The system can set all groups to the diagnostic mode with an input/output command. In this mode, commands are available to read the registers, and to set external sync. In addition, the system can simulate the interrupt on input, using diagnostic commands. The system can test the input/output, and interrupt functions of the interface in the diagnostic mode. Notice that this procedure permits isolation between the processor, the interface, and the external device so the user can quickly and efficiently isolate the source of trouble in the system. This capability is essential in practical applications. The integrated, digital input/output interface permits flexible connection of any input/output device with the processor provided the device speed of response is compatible with: 1) the processor response time to interrupt, and 2) the necessarily slow control exercised by programmed sequences of operations.

The Direct Program Control OEM Interface

The input/output channel of the IBM Series/1 processor provides a very general set of commands as discussed in Chapter 5. As OEM devices to be interfaced to the processor become more complex and require a more rapid response time, the previously discussed interfaces become less appropriate. What is needed is an interface which permits all the various input/output commands to be exercised through that interface. Except for that portion used for cycle steal transfers, the interface makes available the entire input/output bus. The direct program control OEM interface provides

a sufficient subset of the input/output bus to enable a user to add hardware to this interface while taking maximum advantage of the input/output channel capability. In addition, the interface provides the previously-discussed self-checking and diagnostic capability.

OEM Interface Architecture

Figure 103 is a block diagram of the OEM interface. As with any interface, data input, data output, and interrupt request lines are provided together with control lines. Unlike the previously discussed interface, the Series/1 does not provide buffer registers. The interface is designed to provide all direct program control functions for up to sixteen devices connected to the buses in Figure 103. The user must provide all the external hardware necessary to connect multiple devices to these lines, including:

- Buffer registers
- Logic to detect which device is addressed
- Logic to control interrupt request and control lines

The interface provides only the bus subset. Diagnostic capability is important in such an interface; without this capability, it would be exceedingly difficult and time consuming to determine on which side of a user-provided interface an error occurred. Self-checking operates within the interface itself. In addition—to provide a thorough check of the interface operation—diagnostic instructions permit data to be transmitted out through the interface, wrapped around, and read back in again. This procedure is discussed further in Chapter 9.

The OEM Interface Bus

The direct program control OEM interface bus contains 75 lines (Figure 104). The functions and need for most of these lines are self-evident. For example, 36 of the lines are grouped into 18 input lines and 18 output lines (all data transfers are 16 bits in length with two parity bits). Since the interface supports up to 16 devices, the system needs four more lines as device address lines. To support all direct program control functions, devices must be able to signal their desire

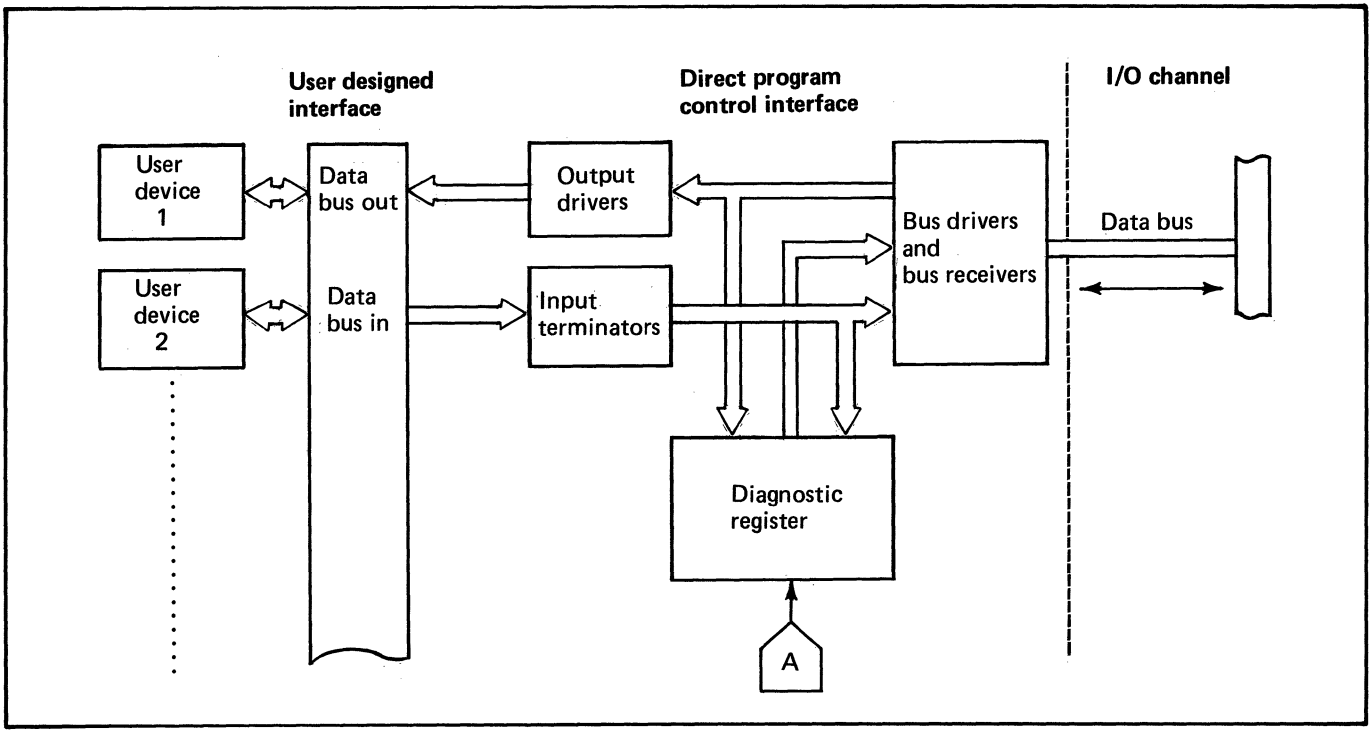


Figure 103. Block diagram of the OEM interface (1 of 3)

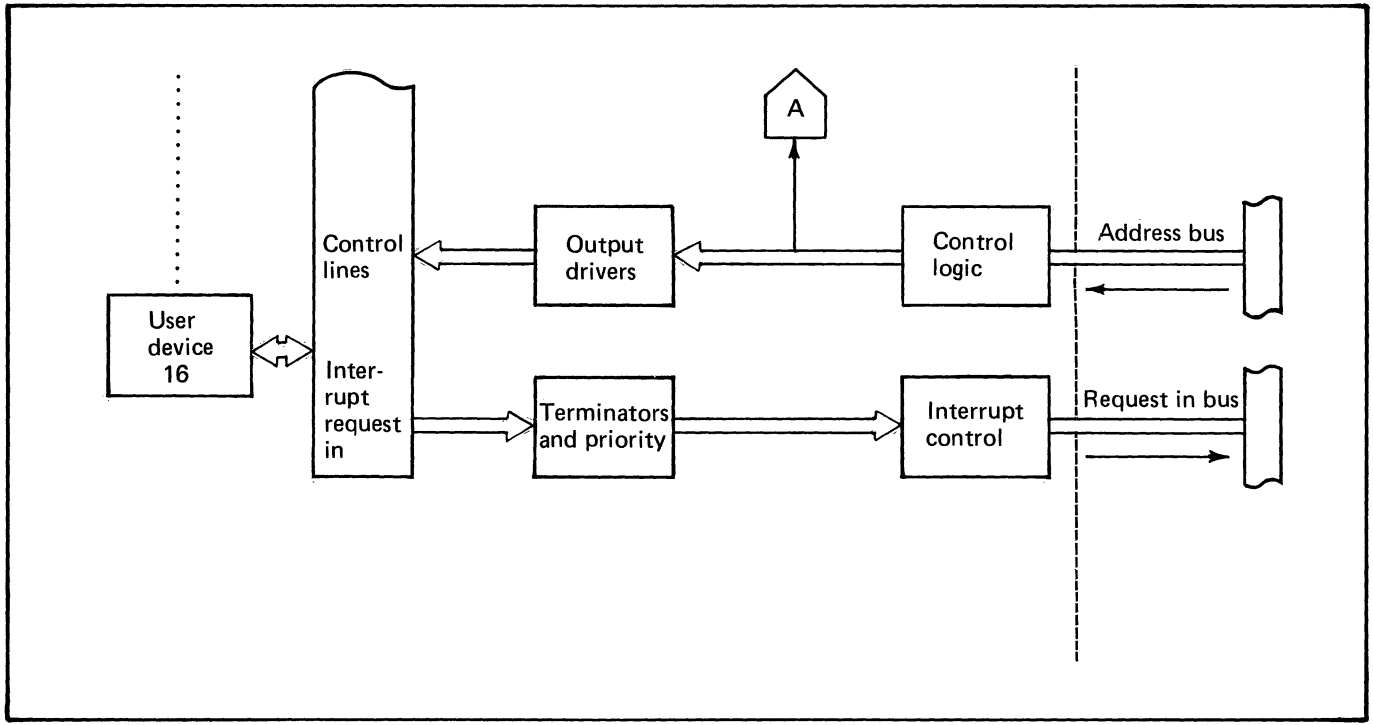


Figure 103. Block diagram of the OEM interface (2 of 3)

User interface functions include:

- Detecting which of 16 devices is addressed
- Buffering all data and addresses
- Controlling handshaking using control signals to and from the direct program control interface
- Generating interrupt signals and condition codes to be consistent with the Series/1 architecture

The direct program control interface permits the connection of up to 16 user-supplied special devices to the Series/1 processor through a subset of the full input/output channel. All input/output transfers are restricted to the direct program control type (one sixteen bit data word transmitted per Operate I/O instruction executed). The interface allows full use of the input/output system for user-supplied devices including self-diagnostic features.

Figure 103. Block diagram of the OEM interface (3 of 3)

to interrupt. Consequently, 16 of the lines are devoted to interrupt request signals from each of the 16 devices (a device signals an interrupt and holds that request line until recognized—hence, the 16 interrupt request signal lines cannot be coded into 4 lines as device addresses can be).

Command Lines

Full support of all input/output commands—including sub-functions specified by modifier bits in the command—requires seven additional lines (as shown in Figure 104) to transmit the function and modifier bits contained in the immediate device control block command field. Notice that it is the responsibility of each device connected to this interface to interpret those fields which require special logic. Clearly, as far as the user's design effort is concerned, this interface is more complex than any discussed previously; this complexity is necessary if the system is to make all input/output channel functions available to the attached devices. When the system accepts an interrupt, the device must supply a condition code which is reported in the level status register—this action requires another three bits on the bus.

Control and Timing Lines

Nine bits remain for control and timing purposes. Five of these nine are used to signal specific modes or commands:

1. System reset
2. Power-on reset
3. Diagnostic mode
4. Diagnostic mode modifier
5. Processor halt

Attached devices must respond to each of these special commands in a standard way. For example, system reset demands that pending interrupts be reset, and registers and buffers be cleared. Refer to the appropriate processor's User's Attachment Manual to define the mandatory device responses to these special commands.

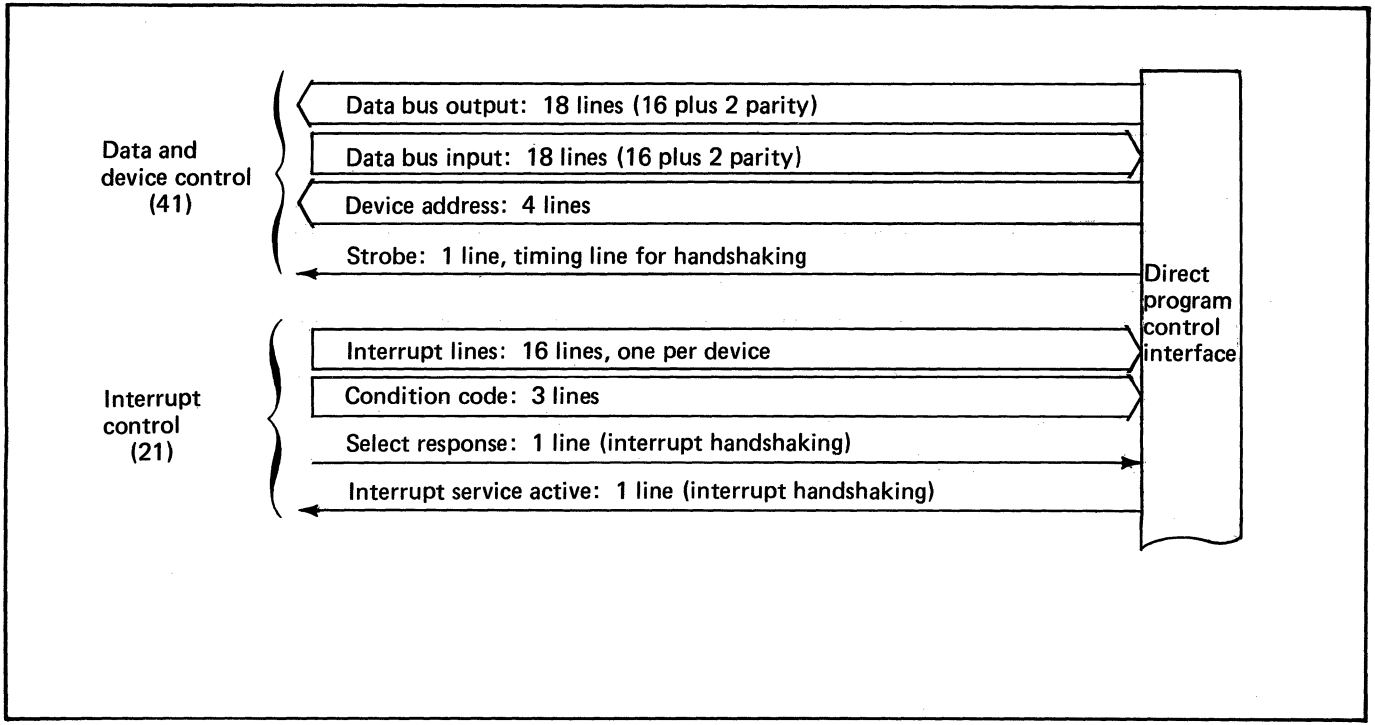
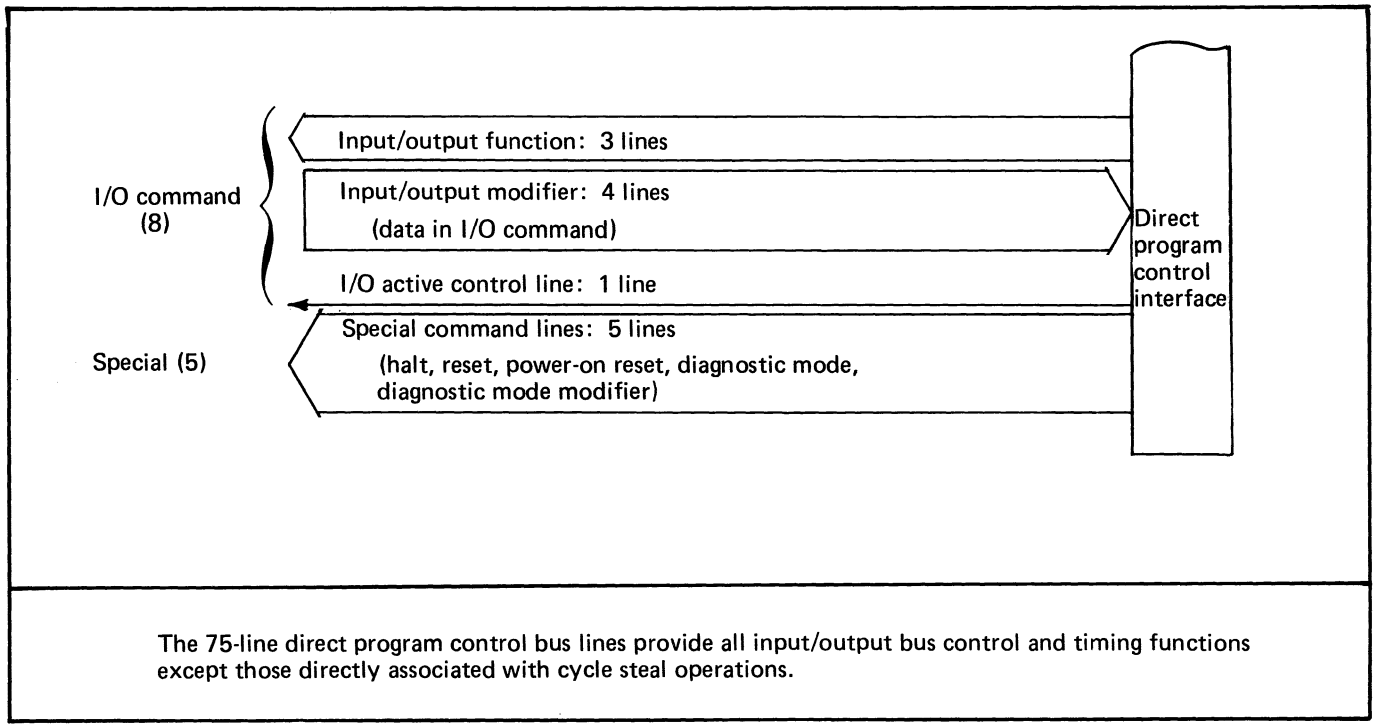


Figure 104. The direct program control interface bus (1 of 2)



The 75-line direct program control bus lines provide all input/output bus control and timing functions except those directly associated with cycle steal operations.

Figure 104. The direct program control interface bus (2 of 2)

Interrupt and Timing

The system uses the remaining four bus lines—I/O active, interrupt service active, strobe, and select response—for timing and handshaking purposes. The *I/O active* line signals that a valid command is present; this means that device address, function, and modifier lines have been written, have settled, and may be read and executed. The *interrupt service active* line signals devices that the interrupt service sequence can begin. Devices previously signaling a request to interrupt can examine device address lines to see if they have been selected, and respond appropriately. *Select response* is a handshake signal from a selected (addressed) device. *Strobe* is a corresponding handshake or timing signal from the processor.

Thus, the system uses most of the 75 lines on the bus to transfer data, addresses, condition codes, and similar information; only a few control timing. For a device connected to the interface, this arrangement greatly simplifies the design of the external hardware. In fact—except for the necessity to provide service for multiple devices and for the full use of all commands—the interface is conceptually similar to the simple, integrated digital input/output interface discussed earlier.

Typical Output Sequence

The architectural simplicity of the interface is illustrated by considering typical input, output, and interrupt sequences. Figure 105 shows the output sequence:

1. The system places the function, modifier, device address bits, and data on their appropriate lines
2. The *I/O active* tag is skewed (at least 200 nanoseconds), and activated on the interface
3. Upon recognition of address compare and *I/O active*, the device raises the *select response* tag. Once raised, the system must hold this tag active at least until the fall of the *I/O active* tag. The device sets *condition code in* which must remain active until *strobe* becomes active, or until *I/O active* becomes inactive (for the duration of the *select response* tag).

4. *Strobe* is activated and dropped
5. The *I/O active* tag is deactivated
6. Upon recognition of the absence of the *I/O active* tag, the device drops *select response* and *condition code in*
7. The system deactivates the function, function modifier, device address, and data busses

Typical Input Sequence

The input sequence is similar (Figure 106):

1. The system places function, modifier, and device address bits on their appropriate lines
2. The *I/O active* tag is skewed (at least 200 nanoseconds), and activated on the interface
3. Upon recognition of address compare and *I/O active*, the device raises the *select response* tag. Once raised, the system must hold this tag active at least until the fall of the *I/O active* tag. *Data bus in* and *condition code in* must be active until *strobe* becomes active, or until *I/O active* becomes inactive (for the duration of the *select response* tag).
4. *Strobe* is activated and dropped
5. The *I/O active* tag is deactivated
6. Upon recognition of the absence of the *I/O active* tag, the device drops *select response, condition code in,* and *data bus in*

Interrupt Response

Each interrupting device has a dedicated line which the device may raise at any time. The system maintains the signal until either the interrupt is accepted or a reset command is received. All devices attached to the interface are prepared with the same command; they are enabled or disabled as a group, and they interrupt on the same hardware priority level. The processor recognizes which device interrupts (priority among simultaneously interrupting devices is from lower- to higher-numbered addresses on the interface).

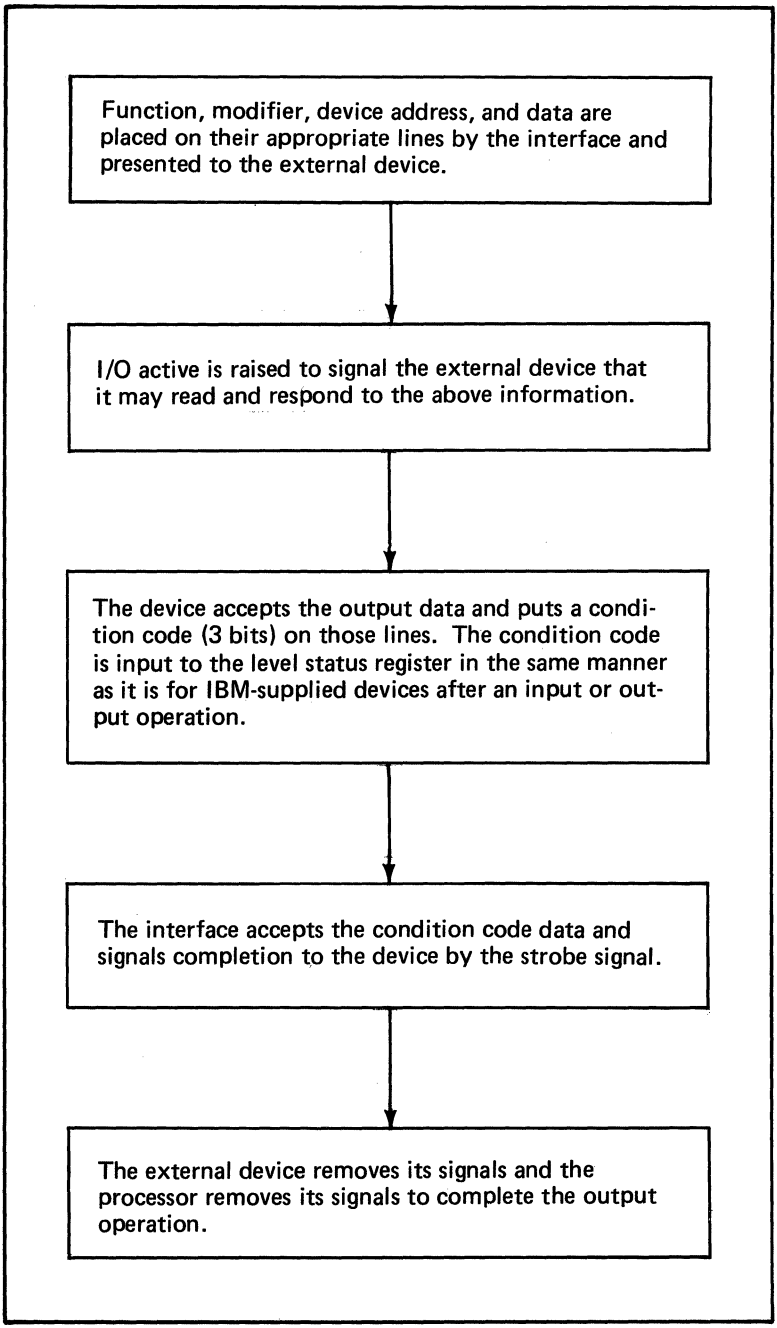
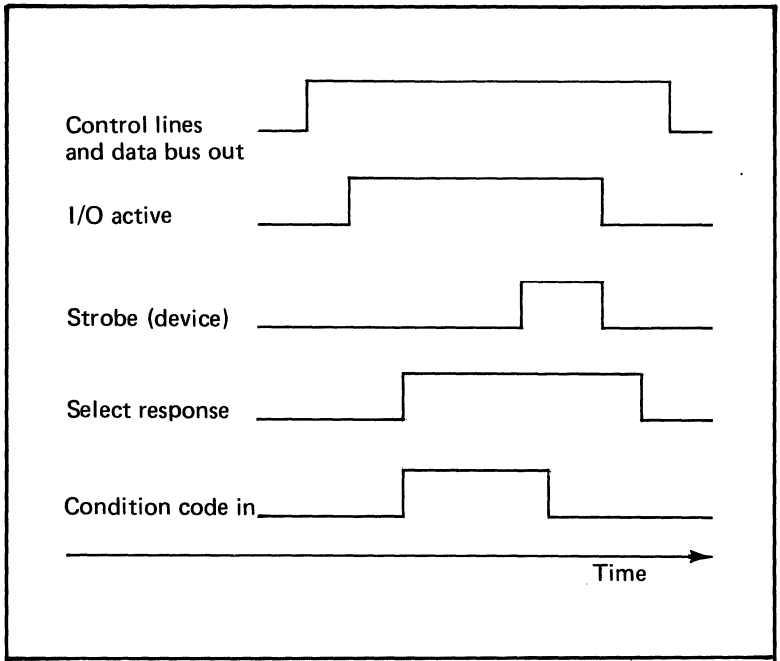


Figure 105. Data bus output sequence (1 of 2)



An output operation on the direct program control interface follows the same conventions as those for IBM-supplied devices—including handshaking of the data transfer, and supplying of a condition code for processor testing.

Figure 105. Data bus output sequence (2 of 2)

The sequence then proceeds as follows (Figure 107):

1. The system places the device address bits on their appropriate lines
2. The *interrupt service active* tag is skewed (at least 200 nanoseconds) and activated on the interface
3. Upon recognition of address compare and *interrupt service active*, the device raises the *select response* tag. Once raised, the system must hold this tag active at least until the fall of the *interrupt service active* tag. *Condition code in* and *data bus in* must be active for the duration of the *select response* tag, or at least remain active until *strobe* becomes active.

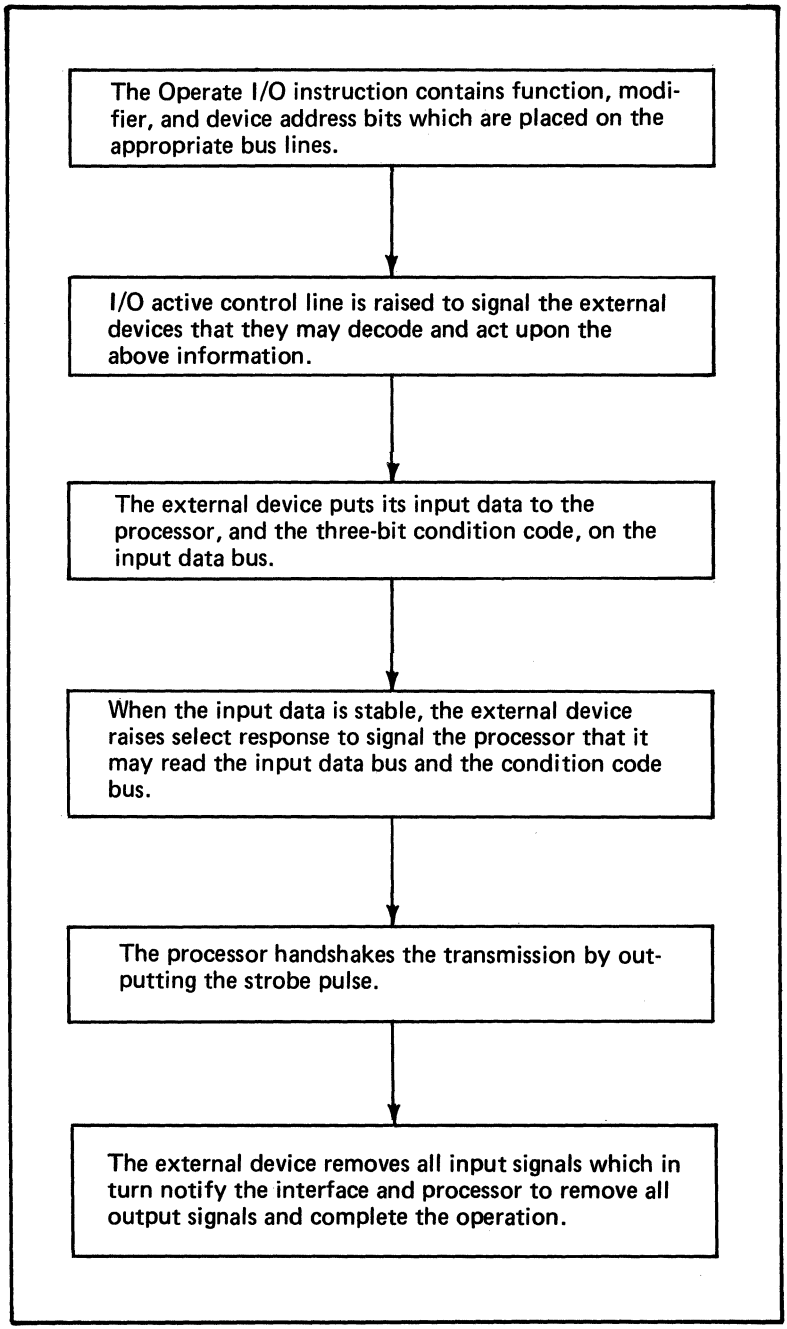
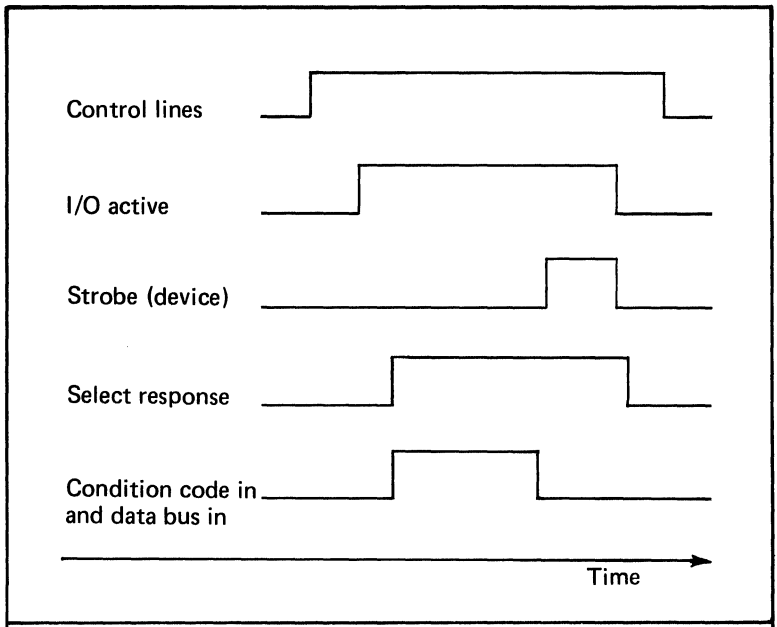


Figure 106. Data bus input sequence (1 of 2)



An input operation on the direct program control interface makes use of all the signals on the bus that are used for an IBM-supplied device. It is the responsibility of the user-supplied external device to detect and respond to all information like command and modifier fields, and to maintain the conventions of handshaking and condition code presentation.

If all conventions are followed, an OEM device—as compared to an IBM-supplied device—may be interfaced without loss of overall system integrity or loss of system software functions.

Figure 106. Data bus input sequence (2 of 2)

4. *Strobe* is activated and dropped. The I/O device must reset its interrupt request at the leading edge of the strobe.
5. The *interrupt service active* tag is deactivated
6. Upon recognition of the absence of the *interrupt service active* tag, the device drops *select response, condition code in,* and *data bus in*
7. The device address is deactivated

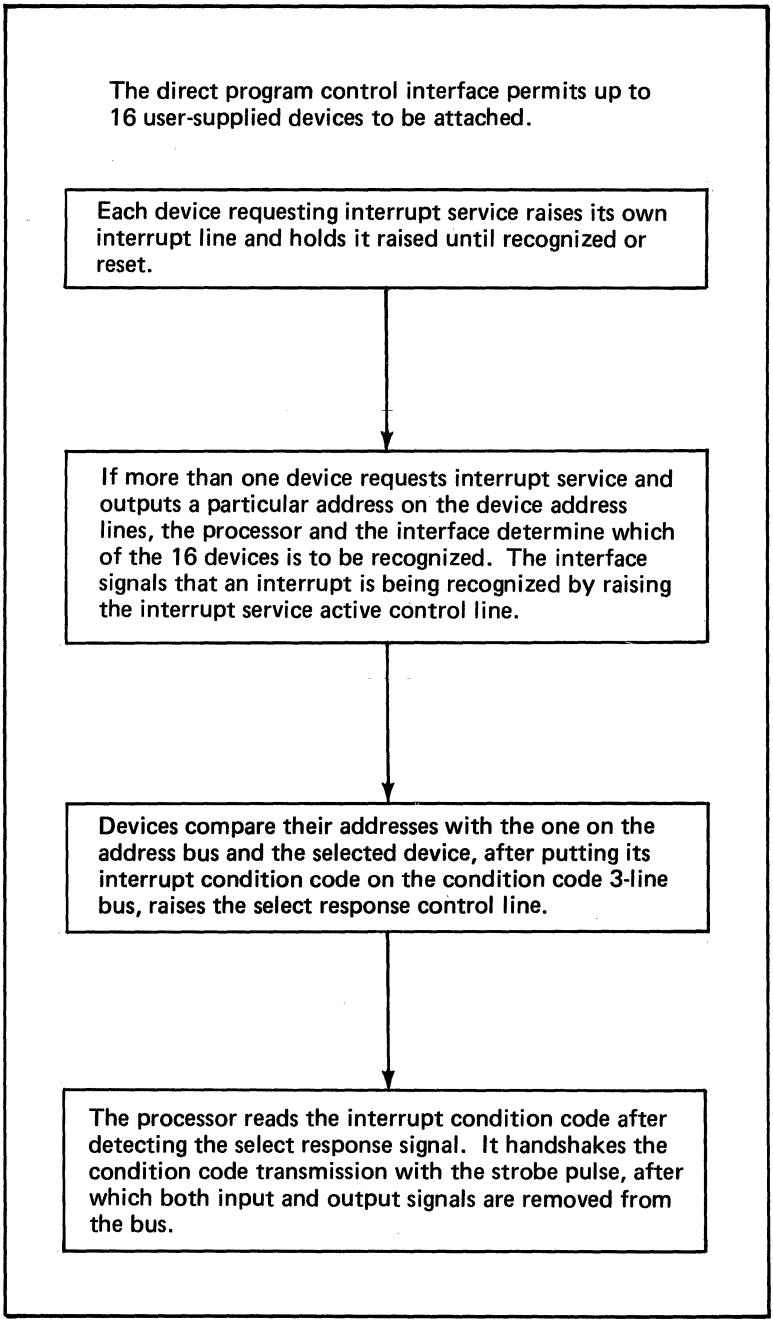
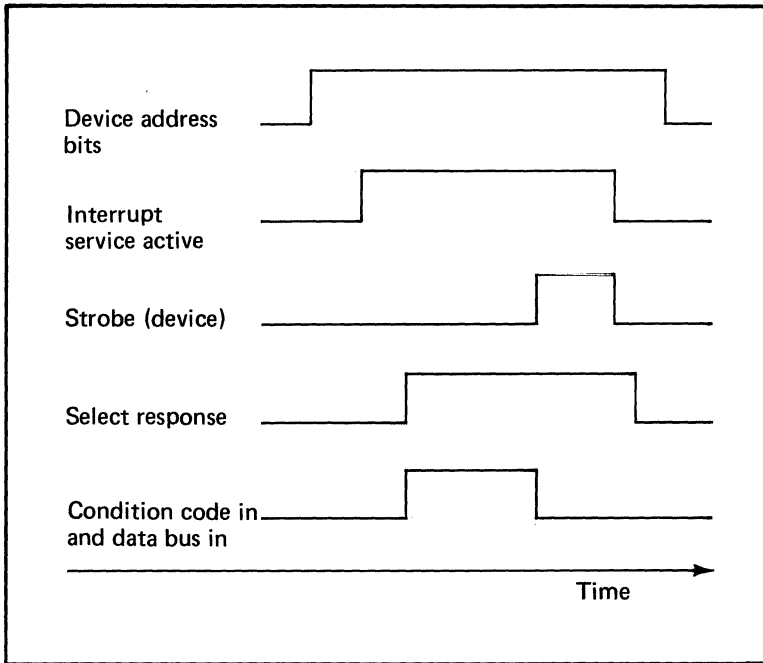


Figure 107. Data bus interrupt sequence (1 of 2)



Individual devices attached to the direct program control interface may interrupt—although all 16 devices are on the same priority level and share a single device interrupt enable flag. Response to the interrupt signals a condition code into the level status register just as it does for IBM-supplied devices. Software handling of the interrupt for user-supplied devices is then identical to that for IBM devices, and is fully compatible with the overall system design. The response to interrupts and transmission of the condition code is fully handshaked.

Figure 107. Data bus interrupt sequence (2 of 2)

Although designing an interface for one or more devices to this bus is not a simple operation, it is not different from any other logical design problem. Users should undertake such a design assignment only for applications which need the quicker response time and greater generality of this interface. Such a design is very economical for applications

which use many devices either on the same processor or on many processors. Obvious examples would be:

- An interface to a distributed, process control data acquisition and direct digital control system
- An interface to machine monitoring and control systems for manufacturing plant control
- Clustered terminals of some special design

OEM users can insure that all functions of the Series/1 can be used, that all functions of their devices can be used, and that overall system integrity cannot be compromised.

Software support of devices attached in this way necessarily depends on the functions the devices perform. Device drivers and interrupt response routines are not conceptually different from those of any standard device; they may be integrated into IBM-supplied software. The user should consult Control Program Support and Realtime Programming System documentation for details of this software interfacing.

Isolated and Directly Connected Channel Interfaces

As long as the direct program control input/output operations are sufficient for control of the user's device, the direct program control OEM interface provides a completely generalized interface. Concurrently, it retains the self-diagnostic capability inherent in the Series/1 interfaces. If the application requires greater speed than the interface can provide, the user may have to perform cycle steal input/output; if so, the interface must be designed to connect as directly as possible with the input/output channel hardware.

Channel Repower

Because most interfaces are neither standard nor similar, the user must assume most of the responsibility for their design. Two aids are provided for this purpose. The channel repower feature is a printed circuit card consisting of IBM

and TTL technology designed to: 1) repower the processor input/output channel signal lines, and 2) provide isolation between input/output card files or user-interfaced devices and the channel. Logically, the repower feature activates the full 81-line input/output channel to allow users to do with it as they please. It is the user's responsibility to respond properly to commands and other system functions: for example, the input/output channel is busy once an operation is initiated—at least until time outs occur. The channel is asynchronous, and waits for handshakes. Hence, failure to provide the correct handshake signals would tie up the channel until monitoring timers take over.

Socket Adapter

The second aid, the socket adapter, is an even simpler interface than the repower feature. It is a card with one connector to plug into the input/output channel backplane connectors; this action directly joins the 81 input/output signal lines to the corresponding leads in a standard connector. In fact, this adapter serves only a single function: to enable standard, printed-circuit card connects to be utilized on the user-designed interface cards. No isolation or electronic capabilities are provided on this adapter. Again, it is the user's responsibility to insure that too heavy an electrical load is not placed on the channel drivers or on any of the other operations performed which might compromise performance of the channel itself.

User-designed interfaces that are attached directly to the input/output channel are normally justified only when a large number are required and their speed is absolutely critical to the application. Sophisticated designers will find that the channel is conventional in its electrical characteristics and, consequently, can be interfaced in a straightforward manner.

Self-Diagnostic Capability

It must be emphasized that the direct channel interfaces do not include a microprocessor-driven internal interface with self-checking and self-diagnostic capability. Therefore,

it is the user's responsibility to insure that system integrity is not compromised. The self-diagnostic capability can be retained if IBM designs a cycle steal interface for the user's device. IBM will consider any user's special order (request for price quotation) for the design of such an interface.

The Instrumentation Interface

One important class of OEM devices often interfaced to computers includes instruments of all types used in:

- Medical laboratories
- Analytical laboratories
- Process and manufacturing control systems
- Research and development laboratories

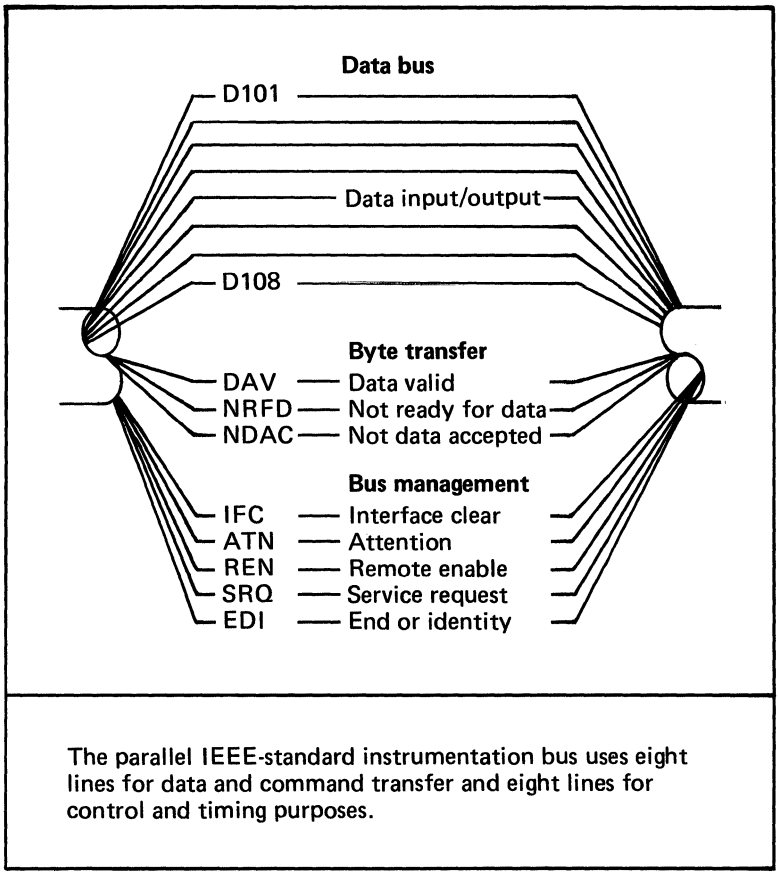
These applications are served by a wide variety of special purpose instrumentation. The data processing industry has long needed a general method of interfacing such instruments to computers for data acquisition and control purposes. The answer to this need was the adoption of a standard sixteen-wire parallel bus called the "Digital Interface for Programmable Instrumentation" in the Institute of Electrical and Electronic Engineers (IEEE) Standard Number 488 (the bus is also an international standard).

The bus devotes eight of its sixteen lines for data transfer (one byte at a time), and eight lines for control purposes (Figure 108). The standard protocol permits:

- Polling of devices
- Communications with one or several devices simultaneously
- Error detection
- Other communications' functions

The design of the bus is especially attractive because of the simplicity of the data transfer mechanism and the consequent ease with which the bus can be included in most new instrumentation.

As an example of the use of the control lines: the data-valid, ready-for-data, and data-accepted lines perform the handshaking functions needed for each byte of information



The parallel IEEE-standard instrumentation bus uses eight lines for data and command transfer and eight lines for control and timing purposes.

Figure 108. The sixteen-line interface bus

transmitted across data lines (Figure 109). The receiver indicates ability to receive data by raising the ready-for-data line. The bus master (or talker) puts data on the data lines and signals its presence by raising the data-valid control line. The receiver accepts the data and then signals the talker by lowering the ready-for-data line and raising the data-accepted line.

The talker then removes the data-valid signal and the data; the receiver removes the data-accepted signal. There may be multiple receivers; the handshake is accomplished by ORing the ready-for-data and data-accepted lines in such a way that the signal is not actually detected on the bus until all

receivers have signals. As a result, the transmission proceeds at the speed of the slowest device involved in the data transfer.

The ORing of signals from multiple devices is accomplished by permitting:

- Each device signaling not ready, to hold the ready-for-data line at ground potential
- Each device signaling ready, to remove the short to ground (Figure 110)

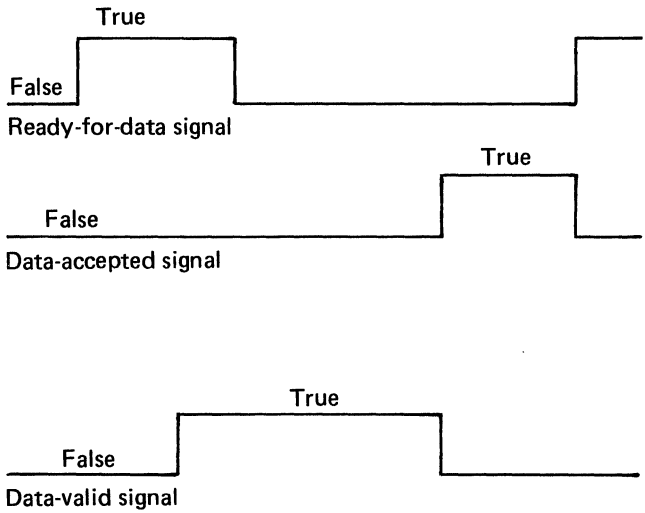
Any one device signaling not ready leaves the bus at the ground level. *Only* when all devices signal ready is there no longer a short to ground; consequently, the bus level rises from ground level only when all devices have signaled ready.

Other control lines are important in setting up the current bus master (or talker) and the current listeners.

For example, the attention control line signals all devices on the bus to watch for their address. This recognition of its own address signals that the device has been selected. Dedication of control lines to specific functions simplifies the design and implementation of the interface for each device. This design simplicity has been the major reason why manufacturers have included an interface for this bus in much of the new instrumentation developed in the last few years.

The IBM Series/1 GPIB Adapter is an interface which couples the Series/1 to this general purpose parallel instrumentation bus. The connection is through the cycle steal storage channel; the data transfer rate to or from main storage can be as high as 65K bytes per second. Data transfer is asynchronous. Direct cable lengths are limited to 20 meters in length.

Like all Series/1 interfaces, the GPIB Adapter is thoroughly integrated into the Series/1 architecture. Utilizing read data/write data level commands, the adapter's microprocessor manages the IEEE interface protocols without CPU intervention. Not only does it fully utilize the cycle steal input/output system for data transfer, but the adapter also extends all of its self-diagnostic capability throughout the interface. That is, the interface itself contains a microprocessor which performs self-diagnosis of the adapter.



1. The talker puts data on data lines and signals data-valid after receiving the ready-for-data signal.
2. The listener detects data-valid and resets the ready-for-data signal while tasking the data.
3. The listener completes the handshake by signaling that data has been accepted with the data-accepted signal.
4. The talker can remove the data-valid signal when it is perceived that the data has been accepted.
5. The listener initiates another cycle, when ready, by asserting the ready-for-data signal.

Each data transfer on the parallel bus is asynchronous and is handshaked; consequently, each step in the transfer is acknowledged by the talker and the listener involved in the operation.

Figure 109. Data transfer coordination

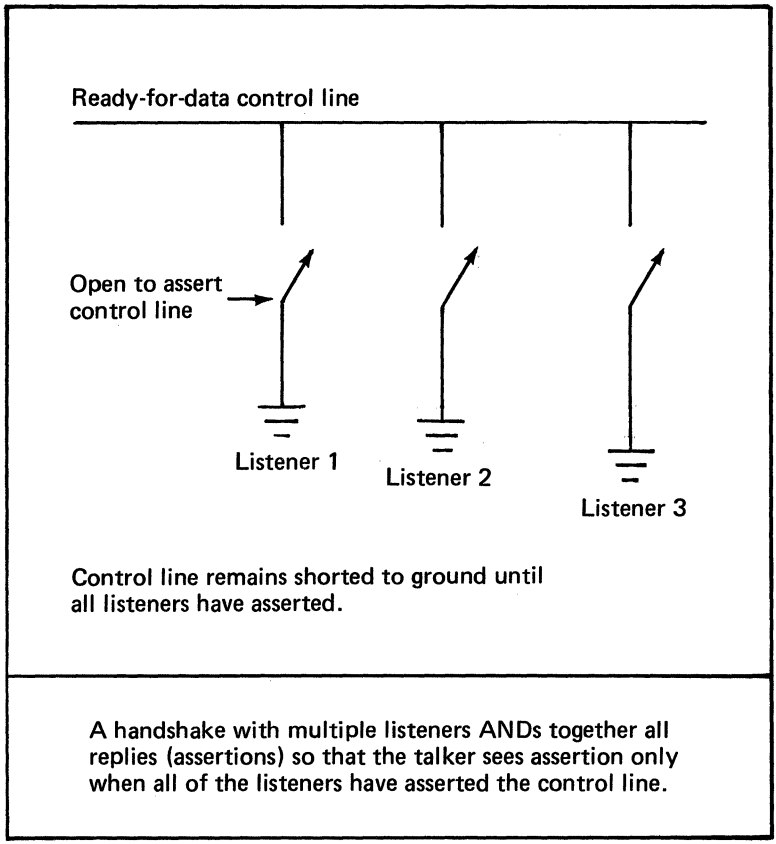


Figure 110. Data transfers with multiple listeners

Upon software command, the interface can wrap back internally, thereby testing that inputs and outputs are being received and transmitted correctly to various points on the system. This process tests the system out to the bus interface itself.

Finally, the sixteen-line cable itself may be wrapped back so that the complete system can be tested—except for the OEM instrumentation connected to the bus. If the system passes all of these tests, signals are being correctly transmitted along the bus and are being correctly received from the bus.

The GPIB Adapter, then, is a very attractive device for connecting instrumentation to the Series/1. The adapter

extends the use of the Series/1 to laboratory-type applications without either sacrificing the architectural integrity of the system or requiring excessive special purpose interfacing.

To summarize the user device interface discussion: there is a wide range of such interfaces available, from standard-device interfaces to completely do-it-yourself types, each with corresponding advantages and disadvantages. This diversity of interfaces permits the user to integrate the Series/1 into almost any type of system for almost any type of small computer application.

8 Distributed Processing Support

The Many Forms of Distributed Processing

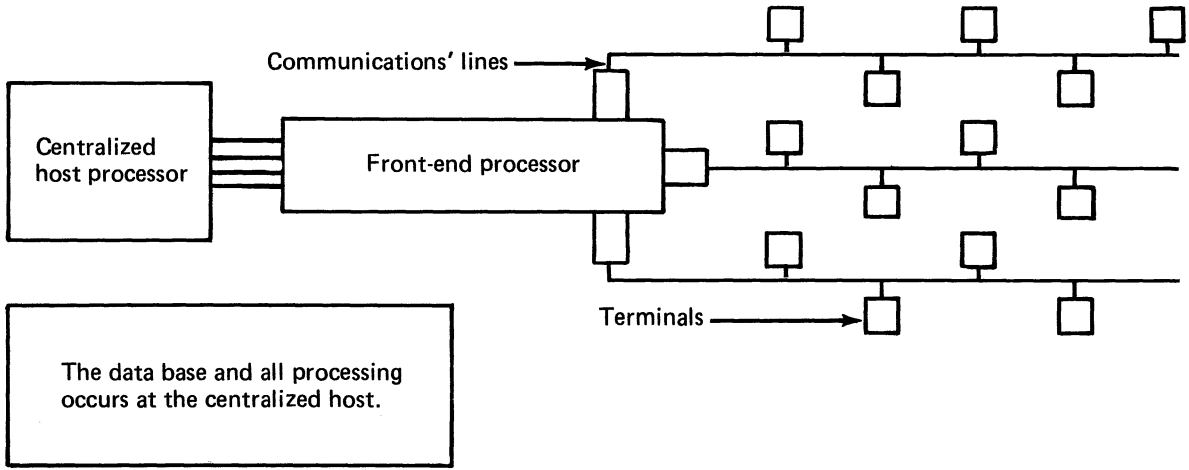
A variety of applications is characterized under the title of distributed processing, including:

- Remote job entry
- Remote interactive data entry
- Remote processing
- Remote data base creation and access

All of these applications require extensive hardware and software communications' support but the structures of the systems can vary considerably. In general, three levels of distributed systems are considered for these applications as shown in Figures 111, 112, and 113.

Centralized Host

The first, simplest, and oldest level uses centralized processing and involves a central host, possibly a small front-end processor, and communications' lines connected to remote terminals (Figure 111). The front-end processor removes the load from the host computer; the terminals then perform as if they were central, relative to the host. Remote job entry and interactive data entry are common commercial applications of this configuration. In data entry, for example, an



The simplest level of distributed processing is a centralized host (possibly with a small computer as a front-end processor) and distributed terminals.

Figure 111. Centralized processing

operator might signal for a specific transaction to be performed. The system codes information into a message and transmits it to the host which then activates an application task to handle that transaction.

The interaction continues with the host transmitting a template-like form which the CRT displays and into which the operator enters and edits the data associated with the transaction. After the operator completes this process, the system transmits the data to the host which acknowledges receipt (perhaps after checking the data). The operator then continues with the next desired transaction. Such distributed systems have proved to be very economical and effective because the terminals can be conveniently located near the source or users of the data.

Remote Processors

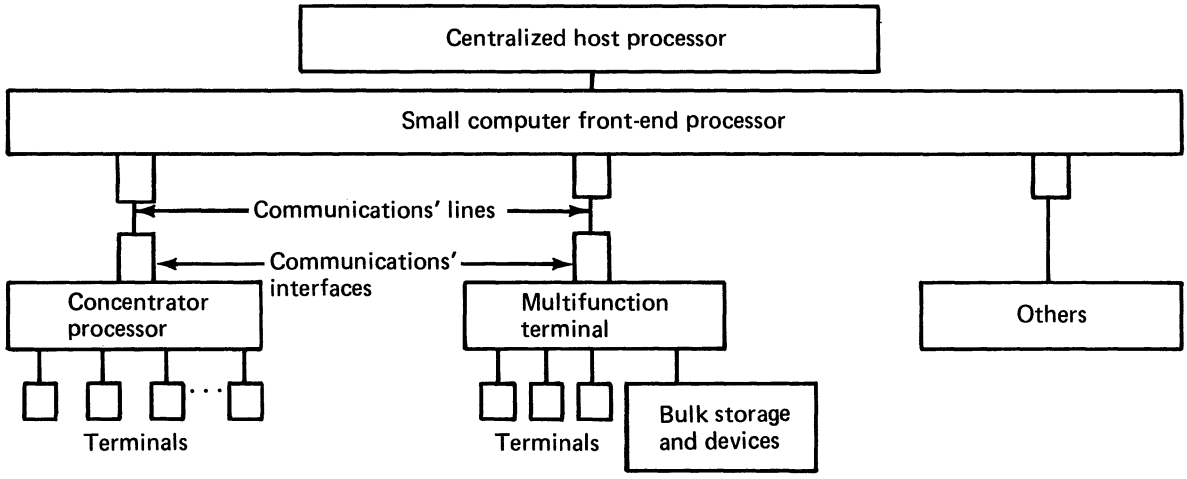
Small computers have made the second level of distributed systems economical for and attractive to users. Figure 112 shows a hierarchical, distributed processing structure where small computers are used remotely to interface and control batch and interactive terminals as well as to do some of the processing. Without host processor interaction, the small computer may:

- Interact in a data entry situation with the complete setting up of a template-form on the CRT terminal
- Interact with the operator during data entry
- Edit and validity check the data and, perhaps, maintain a local data base

These capabilities both off-load the host or front-end processor as well as increase the interaction speed at the remote location. The local processing may be extensive and may be the kind of processing that dramatically decreases the load on the communications' network and the host, as discussed in Chapter 1.

Distributed Application Example

An example of distributed processing that is widely implemented today is the problem of generating a payroll for an



Local data bases may be distributed. Much processing of input data and local data base accessing is distributed.

The second level of distributed processing is the hierarchical distributed system with remote multifunction processors or concentrators.

Figure 112. Remote processing

organization with plant sites remote from the central accounting group, and with differing union contracts. Often these union contracts call for incentive pay scales which depend upon the day-to-day functions that a specific individual performs. In central host machines, payroll-package software is effective in performing the complex functions involved in taking an individual's gross pay and, after considering many taxes and deductions, calculating a net pay. The calculation of gross pay may involve knowledge of detailed production information and may vary from plant to plant. It is convenient to use a small, remote computer:

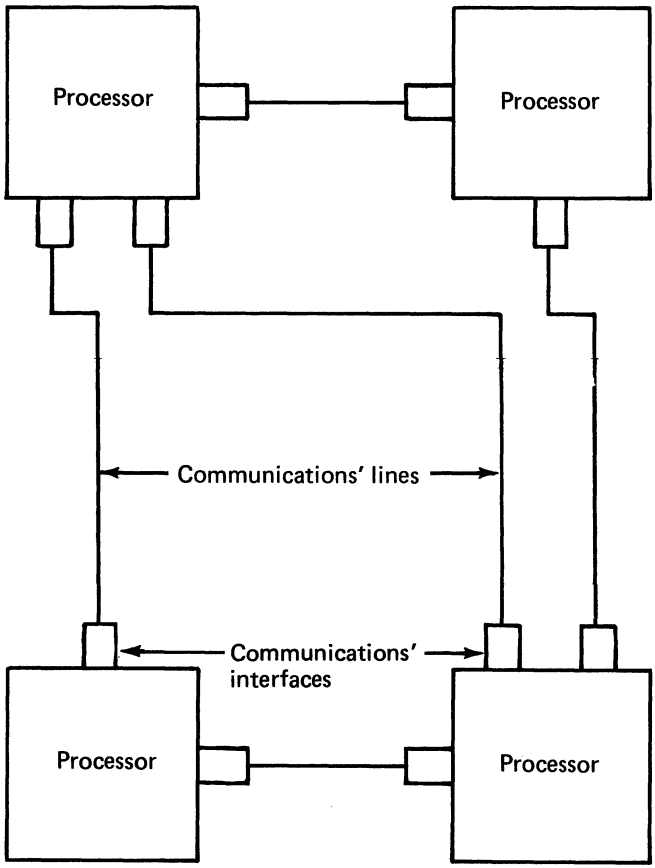
- To gather production information
- To keep the information in a data base
- To generate the gross pay of an individual
- To transmit this data to the central host
  - For use by the payroll package, and
  - For printing the paychecks

The remote small computer facilitates creation and maintenance of the local production data base, including correction of errors. This procedure enables the system to calculate gross pay while using the same data to control production.

The advantages of the hierarchical distributed processing system illustrated in Figure 112 are many; consequently, this level is the most common configuration planned and installed today. Notice that this structure takes advantage of the varied capabilities of the small computer including its higher-level languages, communications' interfaces, and bulk storage, but is dependent for its success upon all the hardware and software needs listed in Chapter 1. A high level of system availability and maintainability is absolutely critical. Hence, the success of the application is very much dependent upon the type of small computer chosen by the user.

Distributed Networks

The third level of distributed processing is a general distributed intelligence network shown in Figure 113. Here, small or large multiple processors may communicate with one



Each processor may act as a host-like processor, a concentrator, or a multifunction terminal. Data base and application processing may be arbitrarily distributed. Each processor must have communications' software to handle message routing from processor to processor in the network.

The third level of distributed processing is a distributed network of processors communicating in an arbitrary way across a variety of links with distributed processing and distributed data base.

Figure 113. A network of processors

another. This system distributes processing, the data base, and users in a rather arbitrary way. Third-structure systems show much promise for the future but are probably several years away from widespread use. Small computers will play an important part in realizing such networks provided their hardware and software architecture is sufficiently generalized. Distributed processing applications may use such a network for communications' purposes—in which case the actual structure and protocols of the network itself will be transparent to users. Networks of small computers may be used for applications like process control where the entire network—or at least much of it—physically resides within one large plant, and where both the network communications' software and the multifunction terminal type of application software are coresident in the small computers.

To insure that their small computers will be compatible with future developments and applications in distributed processing, users must have access to all of the hardware and software requirements discussed in Chapter 1.

First-Level Protocols

There are several structures for distributed processing, as well as several levels of protocols commonly implemented in communications' applications; it is important to distinguish among them when evaluating communications' support. Figure 114 shows three levels of communications' protocols. The first level protocol is used to transmit messages back and forth between two directly connected physical nodes. This level of protocol is responsible for insuring that messages are transmitted properly from Node 1 in the figure to Node 2, and that Node 2 has received these messages properly. This protocol usually involves:

- Keeping a copy of a transmitted message until its correct receipt has been acknowledged by the second node
- Extensive error detection procedures
- Other records substantiating the transmission

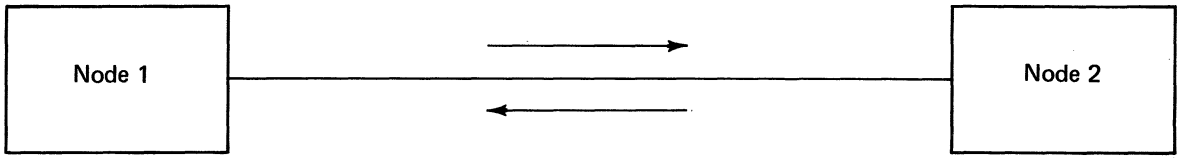
A variety of different protocols is used for this very important level and is summarized in the section of this

chapter entitled "Asynchronous Communications' Protocol and its Hardware and Software Support." Notice that many common applications need this level when two computers are communicating together, or a remote terminal and a computer are communicating between themselves. Integrating this level of protocol into hardware and software is most important. As discussed in the remainder of this chapter, the Series/1 provides extensive support at this level of protocol.

Second-Level Protocols

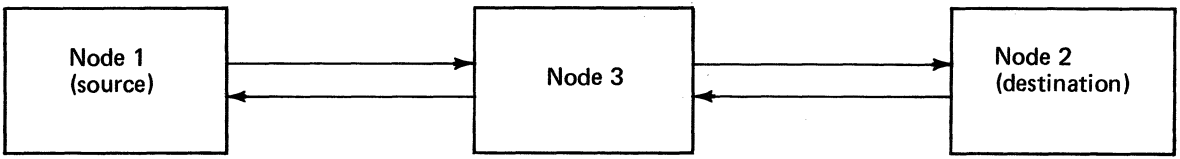
The second level of protocol in Figure 114 involves the exchange of messages between two nodes which are not directly connected. This level of protocol is responsible for setting up the linkage between the two communicating nodes and controlling the exchange of messages. It does this by using the first level protocol on each "leg" of the communications' path, as follows: typically, the system transmits a message from Node 1 to Node 2; the second level protocol determines the message content—involving data like sources and destination addresses. As the system passes this message from node to node along the communications' path, the first level protocol insures its correct transmission and reception. This procedure involves imbedding the original message in a new message which obeys the first level protocol. Hence, the original message is the data portion of the first level protocol message; the system transmits this data portion between pairs of adjacent nodes.

This hierarchical structuring of communications' protocols separates functions and permits a more orderly generation of communications' network systems. Notice that the second level of protocol is not needed in the more common distributed processing applications where the communicating terminals and computers are arranged in a "star" configuration. Most of these applications communicate only from the central node (usually the host) to each individual remote terminal and computer; consequently, the system need not relay the messages across several nodes. The advantage of the hierarchical structuring of the communications' protocols



Level 1

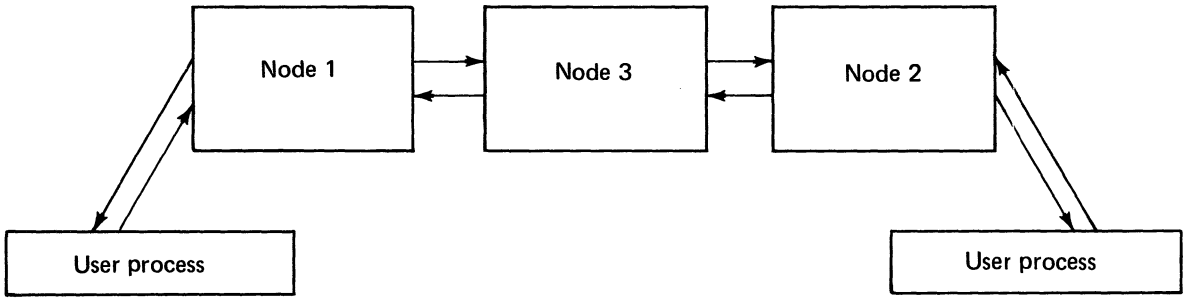
The first and lowest communications' protocol level is responsible for transmitting no-error messages between adjacent nodes. The system performs error detection on each received message and uses retransmission for error correction.



Level 2

The second communications' protocol level is responsible for transmitting no-error messages from a source node to a destination node. As the system passes this message from node to node, it is imbedded into the first level protocol and transmitted to the next adjacent node with no errors.

Figure 114. The three communications' protocol levels (1 of 2)



Level 3

The third and highest communications' protocol level is responsible for the orderly management of a "conversation" or sequence of message exchanges between two tasks. It uses the second level protocol to send error-free messages in both directions between source nodes. The second level protocol, in turn, uses the first level protocol between adjacent nodes.

Today, three communications' protocol levels are operative ranging from the simple first level used in most distributed processing applications to the three level system used in large computer networks.

Figure 114. The three communications' protocol levels (2 of 2)

is that if the functions change or grow in the future, the second level of protocol can be added to a system without changing the first level. This software advantage is just as important to a user as is a hardware architecture that can absorb future changes in technology.

Third-Level Protocols

The third level of communications' protocol shown in Figure 114 is implemented only in large systems. This level of protocol is responsible for the exchange of messages between individual application tasks in separate processors. It involves a procedure analogous to a telephone call: one task calls or connects to a remote task; the second task answers or agrees to exchange messages in an error free, interactive manner. This third level of communications' protocol uses the second level to perform the actual node-to-node communication of a message which, in turn, uses the first level to insure that messages move between adjacent nodes correctly. The IBM System/370 SNA (system network architecture) is just such a protocol. Small computers are ideal front-end processors or stand-alone node processors for such complex communications' applications. It is important that the protocols be hierarchical so that as applications develop and standards are adopted, systems are not obsoleted.

The heart of all distributed processing applications and the basis of all communications' protocols is the first level where two computers or a computer and a terminal communicate; consequently, it is very important that the communications' hardware be very flexible, general, and integrated into the overall system architecture. Furthermore, the support of such hardware must be integrated into system software. If these requirements are satisfied, a user can develop applications economically and add the more advanced, higher levels of distributed processing configurations and protocols if they become necessary in the future.

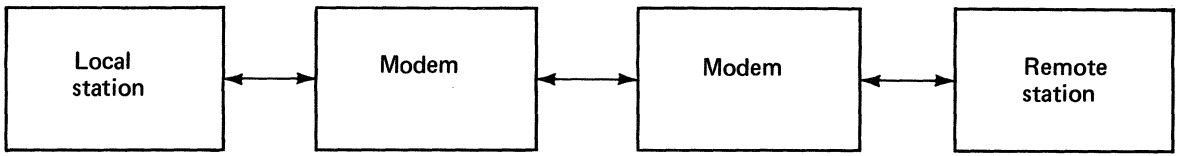
The objective of this chapter is to summarize the communications' hardware and software architecture of the Series/1 in order to demonstrate that it is integrated into the overall

system design previously described, and that it can support distributed processing effectively. The organization of hardware and software for support of communications-oriented applications may vary considerably depending upon the number of communications' lines or terminals involved. Therefore, it is very important to consider the overall hardware/software architecture. This chapter discusses that architecture in detail. Next, the various interfaces appropriate to applications involving a small number of terminals or lines—and their software support—are considered. Finally, the Programmable Communications Subsystem—a microprocessor attachment which facilitates applications involving large numbers of terminals or communications' lines—is discussed. The chapter also reviews the integration of these hardware and software elements.

Structure of Basic Communications' Support of the Series/1

The architecture of the Series/1 is specifically designed so that either simple or complex devices may be connected to its input/output system. Most interfaces are microprocessor controlled to insure self-diagnosis and to provide all the features inherent in the input/output command structure. Complex devices can be processors themselves. This book illustrates that fact in the discussion of the floating-point feature which implements the full set of floating-point operations and conversions. That feature is implemented as a printed circuit card which plugs into the input/output bus. The system implements communications' features in a similar fashion to insure consistency with the overall system design, and to provide the complexity needed to support the variety of different communications' modes and protocols currently used in applications.

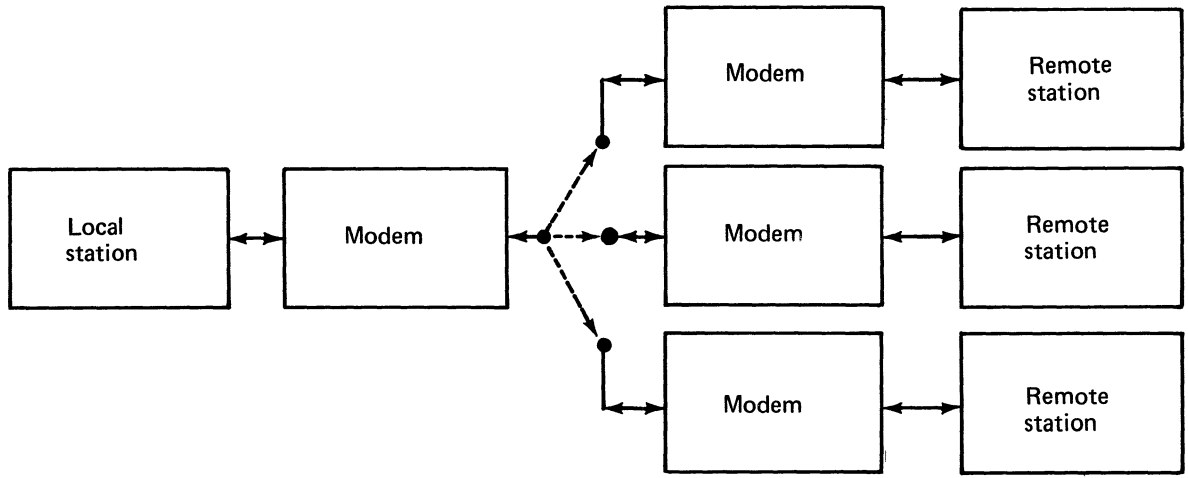
The structure of communications' support is shown in Figure 115 where the Series/1 is the primary station and is connected to remote stations in three different ways. A remote station is either a terminal or another computer. Examples of the latter include another Series/1 processor for



A point-to-point connection of two communicating stations. The modem is necessary only if the distance is long or the telephone communication facilities are used.

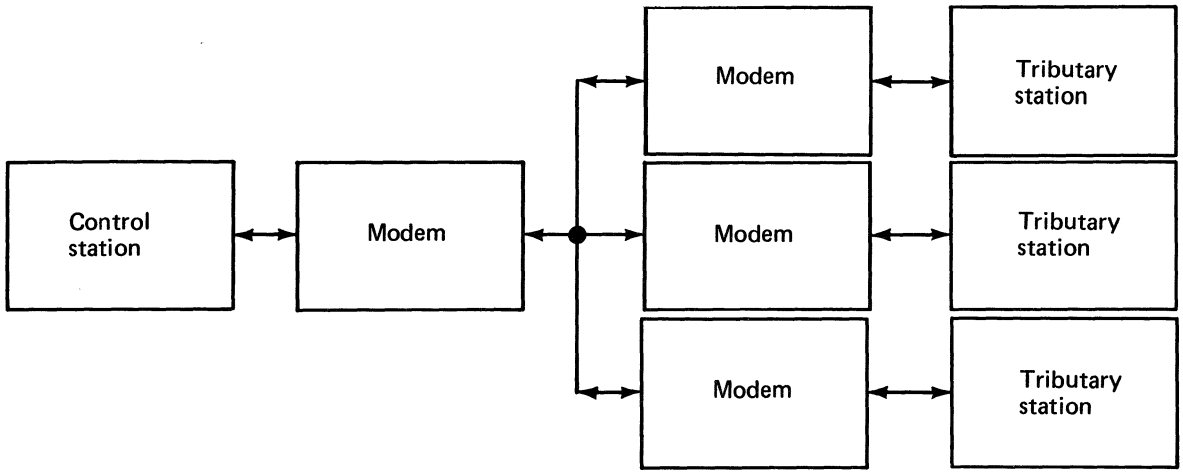
Secondary or remote stations may be connected in several ways, but only one pair of stations communicates at any one point in time. Using a communications' protocol, the local station controls which remote station sends or receives data.

Figure 115. The structure of communications' support (1 of 3)



A point-to-point switched or dial-up connection of communicating stations. Only one secondary station is connected at any time. Both dial-up and responding to the dial-up with Series/1 communications' interfaces can be automatic.

Figure 115. The structure of communications' support (2 of 3)



In multi-point connection of communicating stations, all secondary stations receive the same information across the line or bus. The communications' protocol must call for polling of stations (responding only when addressed) and similar functions so that only one tributary station communicates at a time as directed by the control station.

Figure 115. The structure of communications' support (3 of 3)

small computer to small computer communications, or an IBM System/370 operating under one of its teleprocessing access methods. A terminal consists of a control unit and one or more input/output devices like keyboards and printers. The connection between the Series/1 and the remote station is via communications' lines which may be directly connected to interfaces or driven through modems. The latter are necessary for long distances and across telephone networks to provide adequate signal power and proper signal forms compatible with established systems.

Remote Stations' Connections

Remote stations may be connected in a point to point switched or nonswitched manner (Figure 115, parts 1 and 2). The latter is most commonly used in conjunction with the telephone network which permits connection to a remote station on a temporary basis; the user establishes this connection by literally dialing the number of the remote station. Hardware interfaces, of course, provide the necessary control signals to respond to dial-up connections (for example, an interrupt signal when a ring is detected). In either case, (Figure 115 part 1, or 115 part 2), the communication is between the Series/1 and a single remote station at any instant of time; other remote stations are not connected. Sometimes a "bus" or multipoint arrangement is necessary or desirable as shown in Figure 115, part 3. Here, several remote stations are simultaneously connected to the Series/1. In this case, the system must assign an address to each remote station; it is the responsibility of the primary station—the Series/1—to control which station communicates at any instant of time. Again, only one station communicates at any time; the other stations are connected but must not respond in any way until they are specifically addressed.

Half- and Full-Duplex Communications

Connections between local and remote stations may be half-duplex or full-duplex. In the half-duplex situation, communications can take place in only one direction at a time. In the case of full-duplex, communications can take

place simultaneously in both directions. If only a single communications' path is physically provided, half-duplex operation is necessary. The disadvantages of half-duplex include:

1. Changing of line direction to reverse communications' direction often takes a considerable amount of time relative to the length of transmission (as, for example, when the line must be turned around to transmit a single acknowledgement character)
2. Characters transmitted from a remote CRT cannot be echoed back as received for display on the CRT; instead, they must be displayed as transmitted rather than as received

If two communications' lines are available, they may be used in a half-duplex manner by transmitting on only one line at a time. This has the advantage that all turn around time is eliminated and simplifies the protocols. Full-duplex communications involve more complex protocols such as SDLC to control transmission of messages in two directions—especially when the messages are different lengths, and the system intermixes outgoing messages with those messages which acknowledge receipt of other messages.

Communications' Protocols

The control station governs the communications between the pair of stations by *polling* the remote station: in effect, asking if it wishes to transmit, or commanding it to receive a message. The communications' *protocol* is the convention or agreement on the form which the interchange of information will assume so the stations can understand one another. At the first level, the primary concern is that a message can be transmitted, received, and acknowledged accurately. Depending upon the type of remote station, several different communications' protocols have been developed, and each one is appropriate under different circumstances. Different hardware interfaces are necessary to support each protocol because of:

1. The complexity of these protocols

2. The need to make flexible interfaces which perform as much of the routine processing of the communications' data as possible

Three protocols are used: asynchronous communications, binary synchronous communications, and synchronous data link control (SDLC) communications. All three protocols are supported with hardware interfaces on the Series/1. The following section describes briefly:

1. The protocols themselves
2. The hardware interfaces and their capabilities

Since it is the responsibility of the communications' protocol at the first level to guarantee correct transmission of information, it is necessary that the system provide the receiving station with some means to check for errors, and then inform the sending station whether the message was received correctly or not. As a result, the following general functions must be provided by the communications' protocol:

1. Synchronization information—a signal which permits the receiving station to determine the beginning of a message or character; this signal enables both sender and receiver to interpret the bit serial sequence the same way
2. Message component sequencing—agreement on how addresses are to be transmitted; this sequencing enables multiple secondary stations to determine which one is addressed
3. Error detection information—some means by which the receiver may test the received character stream to determine if noise or error has modified the transmitted message
4. Control conventions—special characters or messages to acknowledge correct or incorrect receipt of a message, reset a device, or perform other functions

Vertical and Longitudinal Redundancy Checks

Error detection can be done by vertical redundancy checking which simply provides a parity bit on each character.

When such a bit is available, the procedure permits detection of an error involving the change in any one bit in the character, and some—but not all—errors involving changes to more than one bit. Hardware permits a choice of even or odd parity, under program control, to facilitate connections to the variety of terminal devices available. Vertical redundancy checking is not feasible if eight-bit characters are used, and the code uses all eight bits for information. Errors often occur in bursts; isolated errors of the type detected by parity checking are, consequently, less common. The system accomplishes longitudinal redundancy checking (also called horizontal redundancy checking) by forming a logical check sum of all the characters transmitted in the message. The check sum is transmitted with the message, and compared to the sum recalculated at the receiver. This procedure permits detection of many more combinations of errors than simple parity or vertical redundancy checking.

Cyclic Redundancy Checks

Modern cyclic redundancy checks are used to maximize the number of different errors that the system can detect with a given number of message error check bytes. Cyclic redundancy checking is a version of horizontal checking in which the check character or characters are generated in the following manner: the system takes the remainder after dividing—by a base number—all the serialized bits in a block of data. Based on elegant information theory concepts, cyclic checks guarantee the detection of all burst errors up to a specified size, and a very large percentage of errors beyond that specified size. With two bytes of error detection information appended to each message, the probability of an undetected error becomes minute. If even this error rate is too large for a very critical application, the application software can simply echo messages back to the source and check them there. The most important fact here is that the system performs error recovery at the protocol level without involving the application tasks because, essentially, all errors are detected automatically with vertical or cyclic redundancy checking—that is, checks which are in the

protocol itself and are, consequently, performed by the interface hardware. Hence, response and throughput of the communications' system is maintained at a very high level.

Data Transparency

One other consideration is important in communications' protocols. Sometimes, it is necessary to transmit arbitrary data items which then comprise a stream of arbitrary characters. The difficulty here is that some characters are reserved for special or control purposes (end of text, for example) and—if detected in the data stream—might prematurely terminate the transmission. When arbitrary data is transmitted, the user must adopt some convention to make the data "transparent" to control characters. Each communications' protocol adopts a different solution to this "transparency" problem.

Asynchronous Communications' Protocol and its Hardware and Software Support

Asynchronous communications between two stations involves a sequence of characters which are not synchronized with one another. As described in Chapter 7, this form of communications is also termed start-stop transmission and involves the following conventions:

1. The two logical levels of the communications' line are called "mark" and "space"
2. Between characters, the line is held in the mark condition
3. Each character consists of a start bit followed by eight information bits, followed by either one, one and a half, or two stop bits
4. Bits within a character are synchronous, with the speed between the two communicating stations being agreed upon in advance
5. The start bit is the mark-to-space transition which—when identified by the receiving station—initiates timing for sampling of the information bits
6. Stop bits put the line in the mark condition

Line Turnaround Characters

Since a message is a sequence of characters, it is necessary to transmit one character at a time with the above format. The communications' protocol is associated with the meaning of special characters and the response of the receiving station to them. Different codes are used with different terminals; consequently, no standard protocol is employed. All protocols, however, involve transmission of a sequence of characters. Some protocols have special meaning for the terminal or computer and cause a line direction turnaround. Thus, a command enabling the terminal to transmit characters is a character sequence for a given terminal type. The system concludes this sequence with a control character which causes the line direction to reverse, permitting the terminal to transmit characters. Transmission of characters (data for example) from the terminal concludes with special characters that involve the receiving computer. If they are to be widely applicable, the variations from one code to another and from one terminal to another call for a very flexible, *programmable* asynchronous communications' interface.

Asynchronous Interfaces

Two such interfaces are provided in the Series/1 system. The first, the single-line asynchronous communications' control interface, provides for one half-duplex line operating at speeds up to 9,600 bits per second. The same interface may be used as a primary or secondary station. The interface itself does not recognize station addresses. As a result, the system cannot use this interface as a secondary station on a multipoint line unless the system provides software within the computer or the device to do the address recognition. The second interface provided is the multiple-line asynchronous communications' control interface which is similar to the single-line interface except that the user may connect a maximum of eight lines operating in half-duplex. The maximum bit rate for each connected device is 2,400 bits per second.

The interfaces operate in the cycle steal mode:

- Accepting bit serial start-stop character sequences
- Assembling them into a byte
- Writing the byte directly into Series/1 main storage using the cycle steal capability of the input/output channel

Cycle Steal Capability

This latter capability is an important consideration when multiple lines are connected because each character received steals only one main storage cycle from the processor. The alternative procedure is to interrupt after each character is received and input the character into storage with a direct program control command. The difference between procedures—in processor overhead time—is much more than a factor of ten. For example, ten lines each operating at 9,600 bits per second correspond to slightly less than 10,000 characters per second (one character transfers in one 660 nanosecond cycle on the 4955 processor). On a cycle steal basis, this takes about 7 milliseconds of the processor time or less than one percent overhead. In contrast, an interrupt per character—responded to by a minimal program—would take around 25 microseconds per character or about 25 percent of the processor time: a ratio of more than 30 to 1.

In order to support cycle steal communications, the interface must recognize control characters because they signal line turnaround necessity. The Series/1 asynchronous interfaces provide for two different character codes which are selectable under program control. The system may define and load line control characters into the interface under program control. Thus, the special control characters may vary, depending upon which terminal is communicating, while the Series/1 still handles them automatically under program control. The user can select the bit rate for the terminal under program control. It is a simple operation to connect many different special control character codes to these interfaces at different bit rates while still providing standard software.

Software Control

Like all I/O devices, input/output commands are used to prepare the interface (select interrupt level and enable interrupts), start the operation, reset the interface, and perform other operations. The system transmits control characters and code designation using the device control block—the eight-word data block addressed in the immediate device control block of an input/output cycle steal command. The system transmits data in this block, on a cycle steal basis, to the channel and then loads it into registers. Each incoming character is compared to these control characters to determine whether or not an interrupt should be generated after the character is loaded into storage. In this way, the involvement of the processor is minimized.

Binary Synchronous Communications' Protocols and Support

The binary synchronous communications' protocol is the most common synchronous protocol in use today; the Series/1 fully supports both single and multiple line interfaces and higher-level languages under the Realtime Programming System. Communications involve messages which are composed of a header, a body, and a trailer each of which is several characters in length. Figure 116 shows the basic message structure. The character sequence is transformed to a bit serial sequence and transmitted serially at speeds ranging from 600 bits per second to up to 56,000 bits per second. Because of the synchronous nature of the transmission, throughput and efficiency are very high. Communications are restricted to the half-duplex mode.

Message Structure

Typically, a message starts with one or more synchronization characters (a predefined character(s) which permits the receiver to get into sync: that is, to align its received character boundaries with the actual characters transmitted). Each field begins or ends with special control characters, for

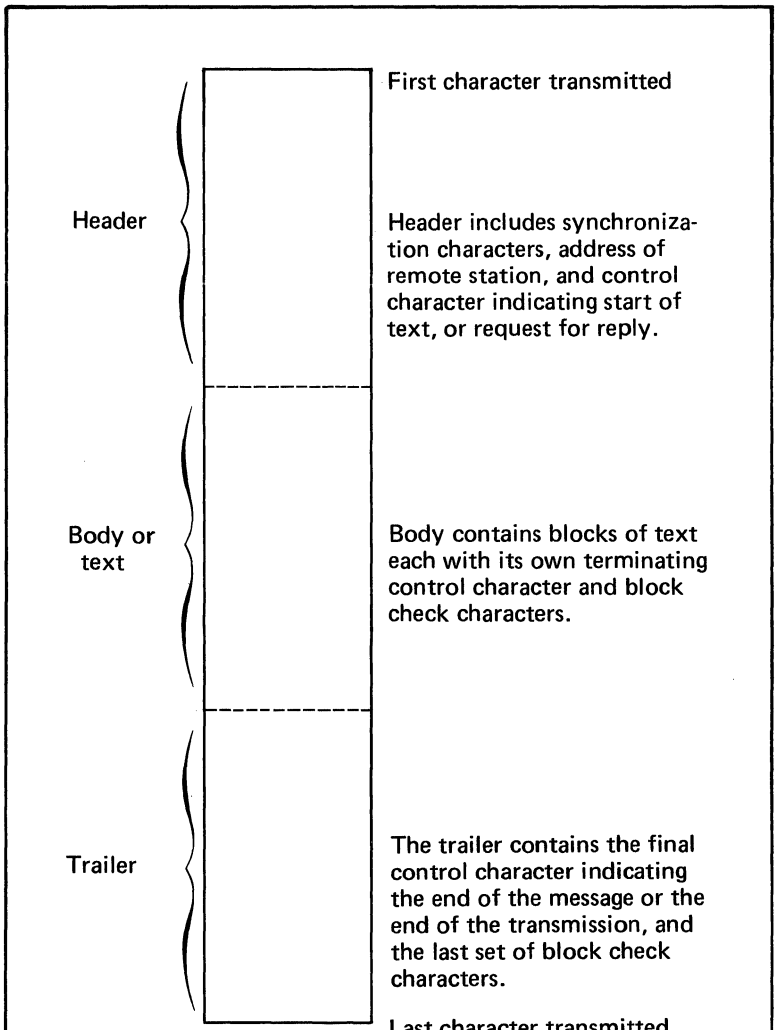


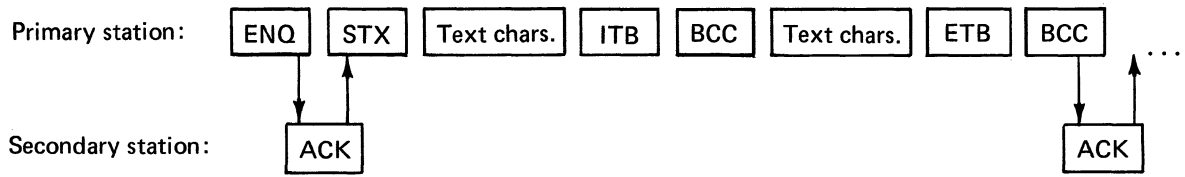
Figure 116. Basic message structure

example start of header, start of test, start of block, end of block, end of test. The message may consist of several blocks, each of which has its own horizontal or cyclic redundancy check characters following the end of block control character. Special control characters are also available for acknowledgement, negative acknowledgement, and similar functions. Designed to operate in the half-duplex mode, each message is acknowledged or negative acknowledged, requiring two line turnarounds for each message.

The system signals transparency by the character sequence DLE STX: two control characters in sequence. Once in the transparent mode, any DLE character which occurs is automatically duplicated on transmission; the system detects the duplication upon reception, deletes the one DLE, and does *not* interpret the other as a special character. Transparent mode is halted by the transmission of a sequence without a duplicated DLE character.

Communications' Example

Figure 117 shows a simple message being transmitted using this protocol. Notice that the system includes two blocks in the single message but acknowledges only the overall message. This procedure reduces the number of line turnarounds that occur, in case of error, at the possible expense of the retransmission of a larger message as illustrated in Figure 118. The illustrated procedure is possible because the system defines a multiplicity of control characters whose meanings are appropriate for different conditions. In this way, end of transmission is different from end of text because it is possible for the secondary station to transmit a reply. In case of error, the receiving station must detect the error during the cyclic or horizontal redundancy check on the incoming data, and inform the transmitting station with another control character. This control character is called negative acknowledge and is used for any "no" answer from the receiving station as shown in Figure 119.



The ACK0 and ACK1 acknowledgement characters are used alternately to signal:

- Ready to receive
- Message received correctly
- Any other "yes" answer

To minimize the possibility of accepting a lengthy erroneous message, the protocol permits multiple breaks—each of which is individually error-checked—in the text portion.

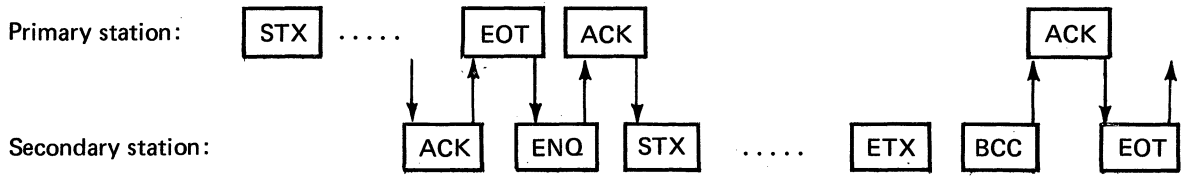
This message contains two text blocks, each with its own error checking characters. The line is not turned around for acknowledgement until the end of text block control character is detected.

The intermediate text block character signals that error checking should be done at that point, but the line is not turned around until the next block. In this way, fewer line turnarounds are required. An error in any block, however, requires all blocks since the last ACK to be retransmitted.

Figure 117. Example of a character sequence for a single message

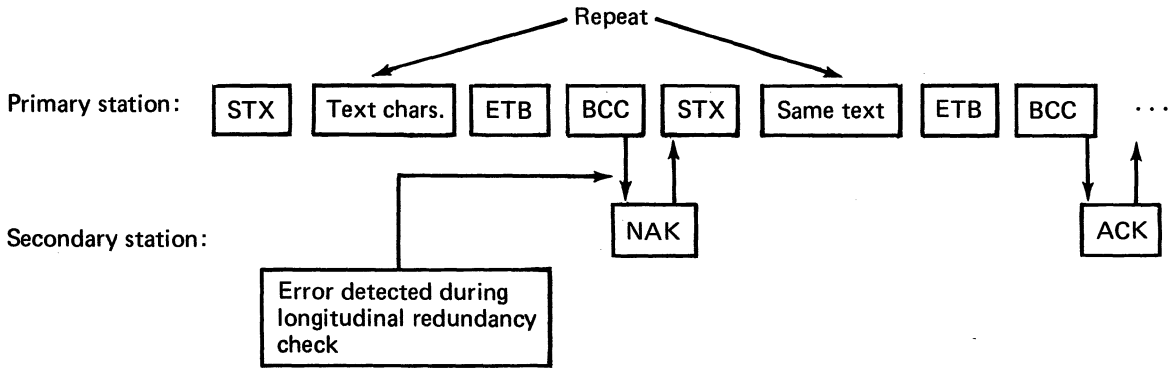
The primary station signals completion of a transmission with an end of transmission control character which turns the line around and permits a reply from the secondary station.

If the secondary station does not wish to reply, it transmits a NAK. Otherwise, it initiates a reply message in the same manner as the primary station.



Messages may alternate in direction at times determined by control characters like end of transmission.

Figure 118. Exchange of messages



Error detection is done with the block check characters and is signaled by a negative acknowledge (NAK) rather than a positive acknowledgement (ACK).

When the receiving station detects an error, the interface hardware performs the block check and detects the error. The station's software turns the line around and transmits a NAK. The primary station then retransmits the entire message.

Figure 119. Error detection

Character Stuffing

For efficiency purposes, the checking of characters for a DLE character, the inserting of an extra DLE, and other checking procedures are, normally, interface (hardware) functions. This technique for transparency is called "character stuffing" because the extra character is inserted into the transmitted stream. In contrast, SDLC uses "bit stuffing" for the same purpose.

The Series/1 provides three binary synchronous communications' interfaces:

1. Single-line interface—one half-duplex line operating at speeds up to 9,600 bits per second with initial program load (IPL) capability
2. Single-line high speed interface—one half-duplex line operating at speeds up to 56,000 bits per second with initial program load capability
3. Multiple-line interface—up to eight half-duplex lines operating at an aggregate speed up to 33,600 bits per second. No initial program load capability is provided on this interface.

Interface Code Support

The single and multiple line medium speed interfaces provide standard EIA and CCITT interfaces for connection to common modems. The interfaces support both EBCDIC and ASCII codes which are selectable under program control (initial program load assumes EBCDIC). Depending upon the code selected, the system performs two different types of error detection checking. Since ASCII is a seven bit code, the eighth bit can be used as a parity bit. By using ASCII transmission, checking is done by parity on each byte received. In addition, the system performs a redundancy check (the logical sum of all of the bytes). The EBCDIC code uses all eight bits for data; consequently, no parity bit is available. A more sophisticated error detection is performed in the EBCDIC mode using a cyclic redundancy check, which is a check over all bytes of the message using a "polynomial" error detection procedure. The hardware interface—

depending on whether it is in the ASCII or EBCDIC mode—accumulates the error detection information as each byte is received and, finally, compares them with the transmitted correct values (called block check characters). The interfaces can also perform block checking on intermediate blocks without processor interaction.

The interfaces use the cycle steal input/output channel to input or output two characters at a time—with a possible exception for the last byte. If the system does not achieve synchronization within a reasonable time or does not receive an acknowledgement within another specific interval, the interfaces provide time-out interrupts.

Figure 120 shows the names and functions of special characters interpreted by the Series/1 binary synchronous hardware interfaces. User application software sets up messages as a character sequence in storage, and then initiates the transmission. As discussed later in this chapter, this set up can be done from the assembly language level, the FORTRAN level, or the PL/I level. The latter two levels make transmission of information to a remote terminal essentially the same as transmission to a local device.

Operating Modes

The primary station controls the transmission and reception of messages as well as the selection of stations. Figure 121 shows the various operating modes which the hardware binary synchronous communications' interfaces must interpret. Control mode is the condition for any interface not being communicated with at the moment. In this mode the interface monitors incoming characters until it detects an end-of-transmission character, after which it monitors, twice in succession, for its station address—the duplication of the address is a precaution procedure in case line noise occurs; the precaution procedure is necessary because secondary stations which monitor do not check the station address with cyclic redundancy checking. Any device not selected will realize this when the system detects the start-of-header or start-of-text character. Unselected devices then idle until another end-of-transmission character places them back in the control mode.

Name	Mnemonic	EBCDIC	ASCII	Mnemonic	Function
Start of heading	SOH	SOH	SOH	SOH	Reset control mode and set the adapter to text mode. BCC accumulation starts with the first character after the first SOH or STX.
Start of text	STX	STX	STX	STX	Reset control mode and set the adapter to text mode. BCC accumulation starts with the first character after the first SOH or STX.
End of transmission block (note 1)	ETB	ETB	ETB	ETB	Reset text mode with block check character (BCC) comparison.
End of text (note 1)	ETX	ETX	ETX	ETX	Reset text mode with block check character (BCC) comparison.
End of transmission (note 1)	EOT	EOT	EOT	EOT	End of transmission.
Enquiry (note 1)	ENQ	ENQ	ENQ	ENQ	Reset text mode without BCC transmission and comparison.
Negative acknowledge (note 1)	NAK	NAK	NAK	NAK	Negative response to a request for a reply, or to a block of heading or a block of text in error.

Figure 120. Names and functions of special characters (1 of 4)

Name	Mnemonic	EBCDIC	ASCII	Mnemonic	Function
Synchronous idle	SYN	SYN	SYN	SYN	Transmitted automatically by the adapter to establish and maintain synchronization.
Data link escape	DLE	DLE	DLE	DLE	Alert the adapter to test the next character for a defined control sequence in transparent text mode. In nontransparent text mode, DLE is treated as data.
Intermediate block character	ITB	IUS	US	ITB	Included in the BCC; it causes the BCC to be sent or received.
Initial program load (note 2)	IPL	DC1 DC1 ENQ		IPL	Control characters to initiate an IPL sequence.
Even acknowledge (note 1)	ACK 0	DLE (70)	DLE 0	ACK 0	Indicate affirmative acknowledgement to even blocks.
Odd acknowledge (note 1)	ACK 1	DLE/	DLE 1	ACK 1	Indicate affirmative acknowledgement to odd blocks.
Wait before transmitting positive acknowledgement (note 1)	WACK	DLE,	DLE;	WACK	Indicate a temporary not ready to continue/receive condition.

Figure 120. Names and functions of special characters (2 of 4)

Name	Mnemonic	EBCDIC	ASCII	Mnemonic	Function
Mandatory disconnect (note 1)	DISC	DLE EOT	DLE EOT	DISC	Used on switched communication facilities only, to initiate a disconnect.
Reverse interrupt (note 1)	RVI	DLE@	DLE<	RVI	Reverse direction of data transfer.
Temporary text delay	TTD	STX ENQ	STX ENQ	TTD	Alert the receiving station to a temporary text delay.
Transparent start of text (note 3)	XSTX	DLE STX		XSTX	Turn off control mode and set the adapter to transparent text mode.
Transparent intermediate block (note 3)	XITB	DLE IUS		XITB	Same as ITB, but also turn off transparent text mode.
Transparent end of text (note 3)	XETX	DLE ETX		XETX	Same as ETB or ETX but also turn off transparent mode.
Transparent end of transmission block (note 3)	XETB	DLE ETB		XETB	Same as ETB or ETX but also turn off transparent mode.
Transparent synchronous idle (note 3)	XSYN	DLE SYN		XSYN	Transmitted automatically by the adapter to establish and maintain synchronization in transparent text mode.

Figure 120. Names and functions of special characters (3 of 4)

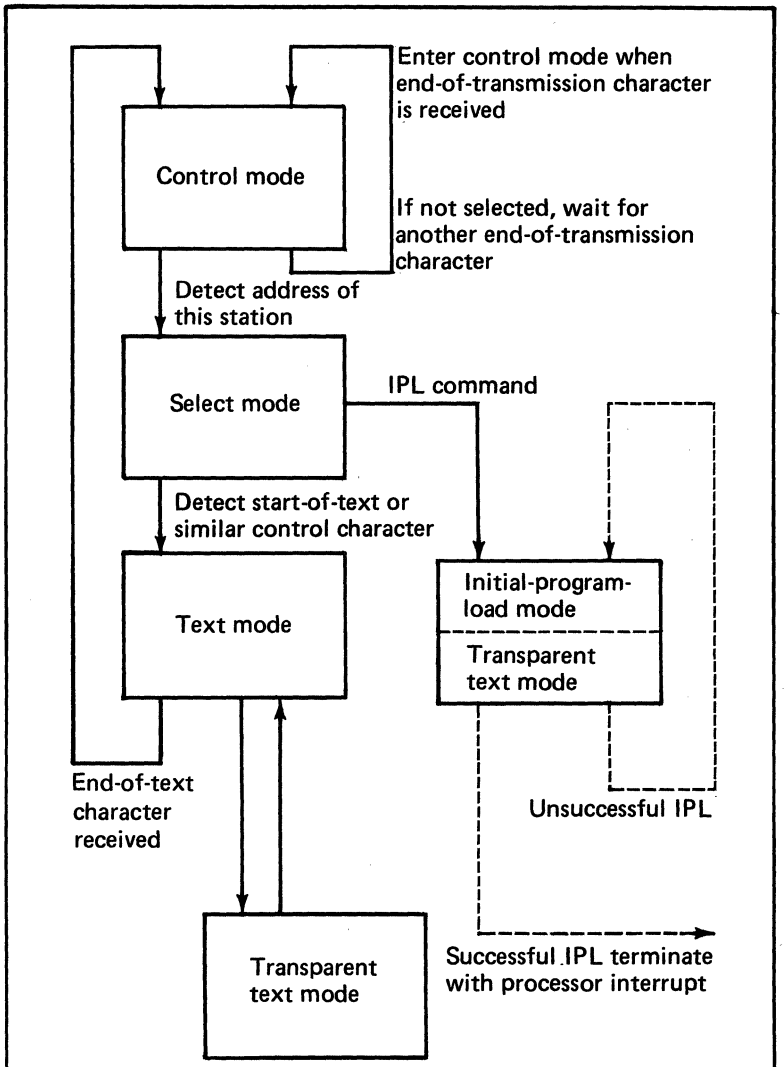
Name	Mnemonic	EBCDIC	ASCII	Mnemonic	Function
Transparent block cancel (note 3)	XENQ	DLE ENQ		XENQ	Turn off transparent text mode and cancel the current block of data.
Transparent TTD (note 3)	XTTD	DLE STX		XTTD	Alert the receiving station to a temporary text delay in transparent text mode.
Data DLE in transparent mode (note 3)	XDLE	DLE DLE		XDLE	In transparent text mode, the transmitter adds a second DLE after each data DLE. At the receiver, the first DLE is stripped off and does not enter storage or the BCC.

Notes:

1. These control characters and sequences cause a COD (change of direction) interrupt request after the required action has been completed.
2. Not applicable in ASCII format.
3. Transparent mode is not available in ASCII.

The binary synchronous communications' protocol is characterized by a number of control characters which have defined meanings and to which connected devices must respond in a predefined manner.

Figure 120. Names and functions of special characters (4 of 4)



The binary synchronous interface operates in one of several modes depending upon whether it is selected by the controlling station for transmission or receiving, for initial program load, or for other functions. The mode changes are caused by detection of pre-defined control characters.

Figure 121. Binary synchronous interface modes

Communications begin once the device recognizes its address. The primary station may transmit a message to the selected device by placing it in text mode and—if necessary—transparent text mode. Certain control characters demand a response from the secondary station; in this case, the half-duplex line turns around. For example: after each text block, the end-of-transmission block character turns the line around so that the receiving terminal can either acknowledge or negative acknowledge depending upon whether or not the system has verified the block control characters.

Control Characters

The system uses control characters to invoke specific responses. For example, an enquiry (ENQ) asks if a remote station wishes to transmit to the local station. This enquiry causes a line turnaround and activates either an acknowledge character (ACK) if the system wishes to transmit or a negative acknowledgement (NAK) if it does not.

Other control character sequences are similarly interpreted as invoking special functions. The system originates the Initial Program Load command as shown in Figure 122. The address of the device on a multipoint line follows the two synchronizing characters. The two DCL characters are, by convention, interpreted as an IPL command. The ENQ character asks if the down-line processor is ready to receive the IPL. The line is turned around and acknowledged with the control character ACK0, as shown. The host then puts the line into transparent mode by transmitting the DLE STX two-character transparency command sequence.

The IPL message is simply the program transmitted a byte at a time (the system must use the transparency mode since the bytes of the program can take on any arbitrary value). The interface duplicates any byte which happens to be the same as DLE and deletes the extra DLE at the receiver.

Finally, transparency mode is left with the DLE ETX sequence in such a way that the transmitting interface does not duplicate the DLE. The receiver leaves transparency mode and acknowledges the IPL message (provided, of course, that it has been received with no error). If the IPL

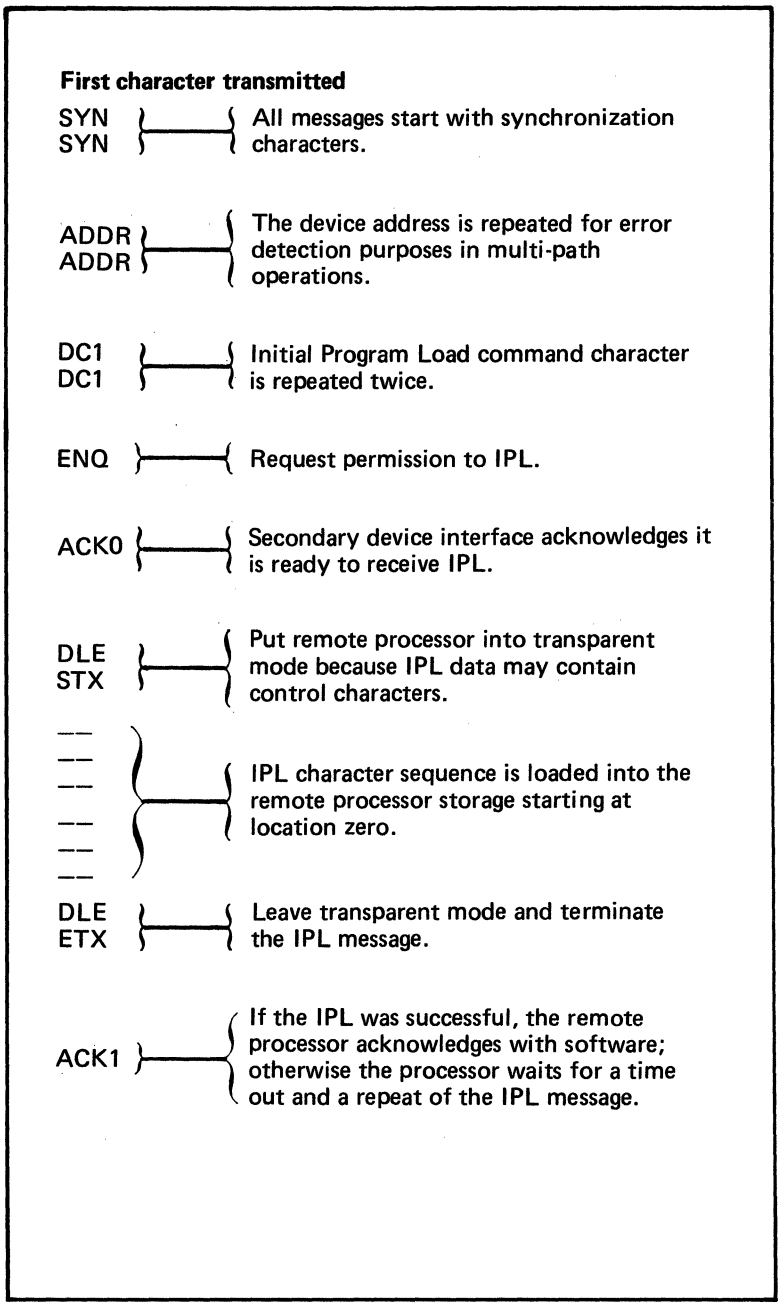


Figure 122. Example of a message exchange containing an Initial Program Load command and acknowledgement

sequence is received correctly, an interrupt occurs in the receiving processor to end the IPL mode. If not, the interface returns to the IPL mode hoping to receive a second IPL sequence. The host processor waits for an acknowledgement of the IPL sequence; if none is received, it reissues the IPL sequence.

The Synchronous Data Link Control Protocol and its Hardware and Software Support

Although the binary synchronous communications' protocol is common and useful, it has one limitation which requires a different protocol: namely, it cannot be used in full-duplex mode. Full-duplex lines may be used to avoid the line turnaround time when simple ACK or NAK one-character messages must be returned; but this procedure does not take advantage of the available line capacity. Full-duplex communications involve sending messages simultaneously in opposite directions. Since these messages are not necessarily either the same size or synchronized in time, there is a complication introduced in acknowledging receipt of correct or incorrect messages: the simple ACK and NAK protocol is not adequate because it does not identify specific messages.

Need for SDLC

IBM introduced the synchronous data link control (SDLC) protocol to handle this problem. It permits full-duplex transmission in a particularly efficient manner. The protocol is rapidly being adopted by other vendors, and is in fact becoming an international standard under the title high level data link control (HDLC). SDLC defines a protocol for communicating an arbitrary message between two nodes (possible on a multipoint line). For communications with IBM devices, IBM has introduced standard definitions for the information part of the message. HDLC provides a unique name to differentiate the general term from the IBM particularization. In general, when HDLC is used to communicate with any device, its manufacturer is free to define the message content in a way meaningful to that device. OEM users of the Series/1 will, of

course, do the same thing when using SDLC to communicate with their devices or processors.

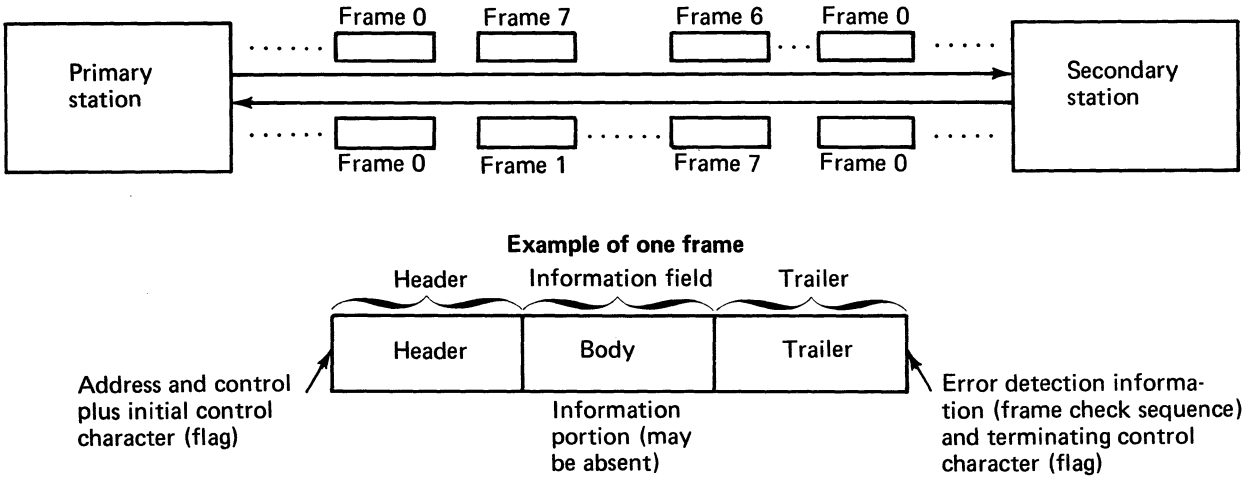
SDLC Messages

Figure 123 shows the basic concept of SDLC communications. Each of the two communicating stations transmits—simultaneously in opposite directions—a sequence of message structures called "frames". The system provides two levels of information grouping for error checking: the frame level and the frame sequence level. As illustrated in Figure 123, each frame contains within it two bytes called the "frame check sequence", which is simply a cyclic redundancy check word for the frame itself. Using these bytes, the system checks each frame received to determine if it was received correctly or not. As a result, just as in binary synchronous communications, the error checking of an individual frame involves: 1) accumulating the check sequence as bytes are assembled from the serial line; 2) finally, comparing the calculated check sequence against the transmitted one.

To acknowledge receipt of a message and to differentiate between frames, each frame is identified with a three-bit number—zero to seven. In the header of each frame is a control field shown in Figure 123 and amplified in Figure 124. The NS three-bit field carries the number of the frame and interprets it as the number of the frame being sent. The system provides the NR field (number received) for the station to acknowledge successful reception of messages. By convention, the receiver keeps track of the number of the next frame to be received.

Message Coordination

Each time the processor or a device receives a frame, it checks the frame first with the error detection word to see if it is correct. If it is, the receiver next checks the NS field of that frame to see if it agrees with the number of the frame the receiver expects to receive. If it does, the frame sequence is correct and the frame is accepted. If the error detection procedures determine that the frame is received incorrectly, it is discarded. If it is received correctly but the



The SDLC full-duplex protocol exchanges messages simultaneously in two directions. The basic unit is called a "frame" which contains frame check sequence characters for cyclic redundancy error detection. Frames are numbered to permit acknowledgement of those received correctly. As many as seven frames may be outstanding before an acknowledgement is required.

Figure 123. Basic concept of SDLC

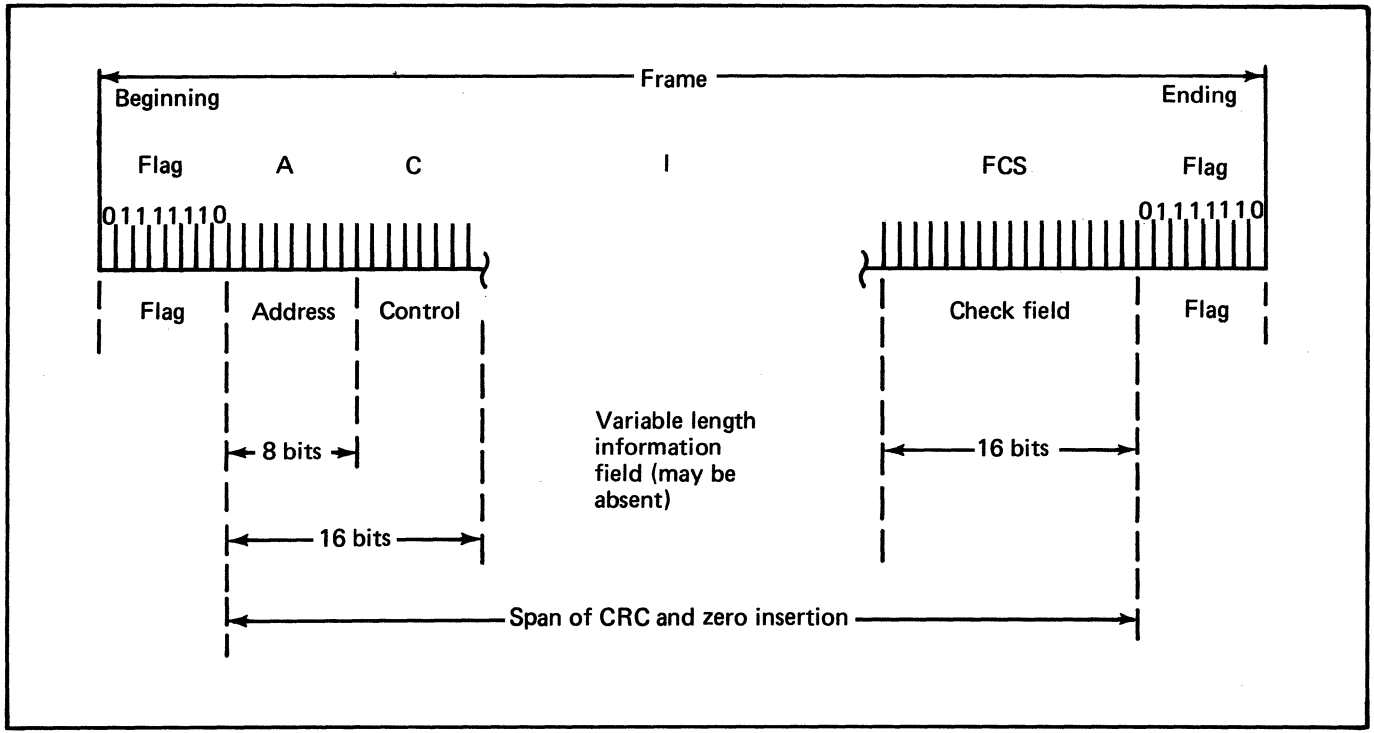
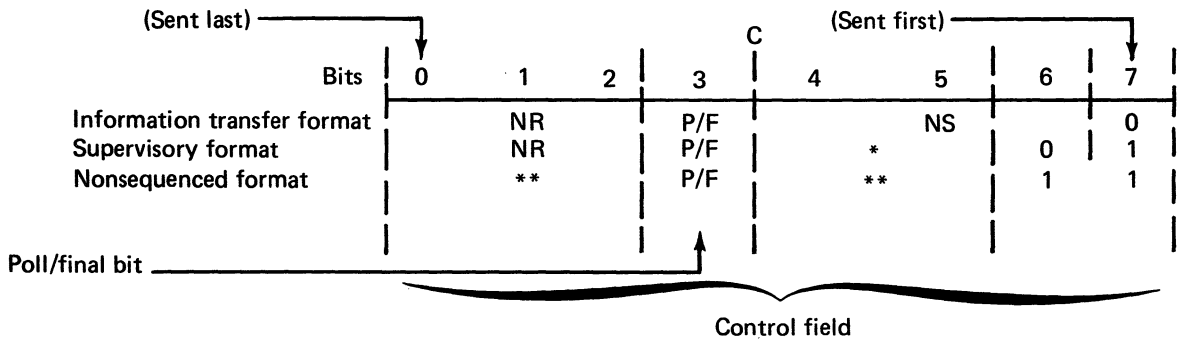


Figure 124. Detailed definition of the SDLC frame format (1 of 2)



One frame is six bytes in length (minimum). Only one control character is used: "flag" which is six sequential one-bits with a zero bit on each end.

The frame check sequence character pair is taken across the entire frame with the exception of the initial and final flag control characters.

The control field includes two three-bit fields; these fields indicate the frame number being transmitted and the frame number next expected to be received. This numbering convention implies that lower numbered frames were received correctly.

Figure 124. Detailed definition of the SDLC frame format (2 of 2)

frame sequence is incorrect, it is usually discarded although the protocol definition does not require this. Whenever the receiving station transmits a frame to the primary station, it carries the current value of NR; that is, the number of the next frame which the receiver expects to get.

When a station receives a frame, it checks the NR field to determine what the receiver expects to get next. Any frame previous to this number in the sequence is assumed to have been received error free; hence, the buffer space in the sending station is freed. The system must retain messages at this point—or beyond—in the sending sequence until a later-received frame acknowledges their receipt. Notice that this procedure allows one station to send more frames than it receives because one frame can acknowledge several messages.

Message Acknowledgement

Because the message count is limited to three bits, a maximum of seven messages may be outstanding before the system requires acknowledgement. That is, once a station has sent seven messages, it must wait until the receiver acknowledges reception of some or all of them. For example: if the receiver reports an NR of 4, messages 4 through 6 must be repeated. If the receiver reports an NR of 7, all seven messages (0 through 6 inclusive) have been received, and message number seven is the next one expected. Notice that even though eight message numbers are defined, only seven messages can be outstanding and unacknowledged at any time if ambiguity is to be avoided.

This fact is illustrated in the above example if the receiver acknowledges with an NR of 0. If the last message sent was number six, this acknowledgement clearly means that no message was received and the entire set must be retransmitted. However, if eight messages were outstanding, an NR of zero could have two references: either to the first message sent, indicating that none were received; or to the next expected message, indicating that all eight were received correctly.

It is the responsibility of user software to generate messages and pass them to the interfaces for transmission. It is the responsibility of the interfaces to handle error detection at the frame level and to notify the processor about control information such as the NR and NS fields.

The examples above were described as if frames always started with zero, but in fact the frame count wraps around with zero following seven. The only frame count restriction is that no more than seven messages starting with any initial count may be simultaneously outstanding without acknowledgement.

With this simple mechanism, SDLC solves the problem of error detection and acknowledgement in a full-duplex environment.

Code Independency

SDLC (and HDLC) have one other very important advantage over the binary synchronous protocol: *the protocol is not code sensitive.* Binary synchronous communications are character oriented with many different characters having a pre-defined meaning as control characters. Hardware interfaces must respond to these characters in different ways. Since devices may use different codes like ASCII and EBCDIC, binary synchronous communications must utilize hardware which is complex enough to handle these codes. Communications need not be complicated this way because—at the first level—the system's objective is to pass messages in an error free mode. Once received, a device or processor may interpret the content of the message in any arbitrary way.

SDLC eliminates code dependence by treating the message to be transmitted as a *bit stream* instead of a character stream. Only one control character is used: the flag, which is a sequence of six one-bits with a zero at each end. Hardware need only count the length of one-bit sequences in order to detect the single control character. As shown in Figures 123 and 124, the start of a frame is recognized by the detection of a flag sequence. The system always treats the next 16 bits as address and control information. The information field is of arbitrary length and may be absent.

Following the information field is the 16-bit frame check sequence, and then a flag sequence. The receiver can tell where the information field ends, or where the frame check sequence begins or ends, simply by detecting the terminating flag sequence. This procedure greatly simplifies the communications' protocol and allows users to assign unrestricted transmission code. The interfaces will:

- Transmit messages correctly
- Break up the bit stream into eight-bit characters
- Pack them into storage

The user program interprets those bytes in whatever way it has been coded to do so.

Bit Stuffing

Since there is only one control sequence—the flag sequence —it is absolutely necessary that it occur in the frame only as the first and last fields; if it appears anywhere else, the receiver would incorrectly interpret its occurrence as the end of the frame. All fields within the frame, except the beginning and ending flag fields, must be "transparent" to flag characters. The particular eight-bit sequence which the system uses as the single control character can occur either as address, control, data, or check information; hence, the protocol must provide for transparency. This is done by "bit stuffing"—an operation that is analogous to "character stuffing" used in the binary synchronous communications' protocol.

Bit stuffing in the transmitting interfaces occurs as shown in Figure 125. Every time the system detects a sequence of five one-bits, it inserts or "stuffs" an extra bit (a zero bit) into the bit stream. For example, if a flag character sequence were to arise in the information field, it would actually be transmitted and received with a zero bit between the fifth and sixth one-bits. At the receiver, the system checks the incoming bit stream. Whenever five sequential one bits followed by a zero are received, the zero bit is deleted.

Figure 125 shows that the transmitter also stuffs the flag sequences. After the system deletes the zero bit following

the five one-bits, the next sequential bit recognized is a one bit. The receiver inserts into storage a flag character occurring within a frame—without the character being interpreted as the end of a frame. The transmitter is responsible for sending an unstuffed flag at the end of the frame. The system performs this transmission easily because the message character sequence is passed to the interface which serializes, bit stuffs, and simultaneously accumulates frame check sequence data for error detection. When the system transmits the last character of the frame to the interface, it appends the two frame check sequence characters and the flag sequence, which end the transmission. Since the interface knows when the end of the frame occurs, it then can prevent bit stuffing in the terminating flag. As a result, the only flag sequence the receiver admits is the terminating flag.

Station Polling

Because communications still occur between a primary and secondary station using the SDLC protocol, polling must still be done. The single P/F (poll/final) bit in the control byte of the frame header is used for this purpose as shown in Figure 126. The polling station sets the poll bit to authorize it to transmit messages. The polled station uses the same bit to accept or reject the invitation. Control over a multipoint line is as orderly as in binary synchronous communications' systems.

The final bit of the control field of a frame was shown as zero in Figure 124. Actually, this bit signals that information transfers of the type discussed above are actually taking place. Notice that the data portion of the frame can be absent if a receiver simply wants to acknowledge a message. This data absence is signaled by the arrival of the normal terminating sequence.

If the final bit of the control field is *one* rather than *zero*, the system interprets the frame as either a supervisory format or nonsequenced format frame. Supervisory format frames are used to convey ready or busy conditions and to report sequence errors. Nonsequenced format frames are

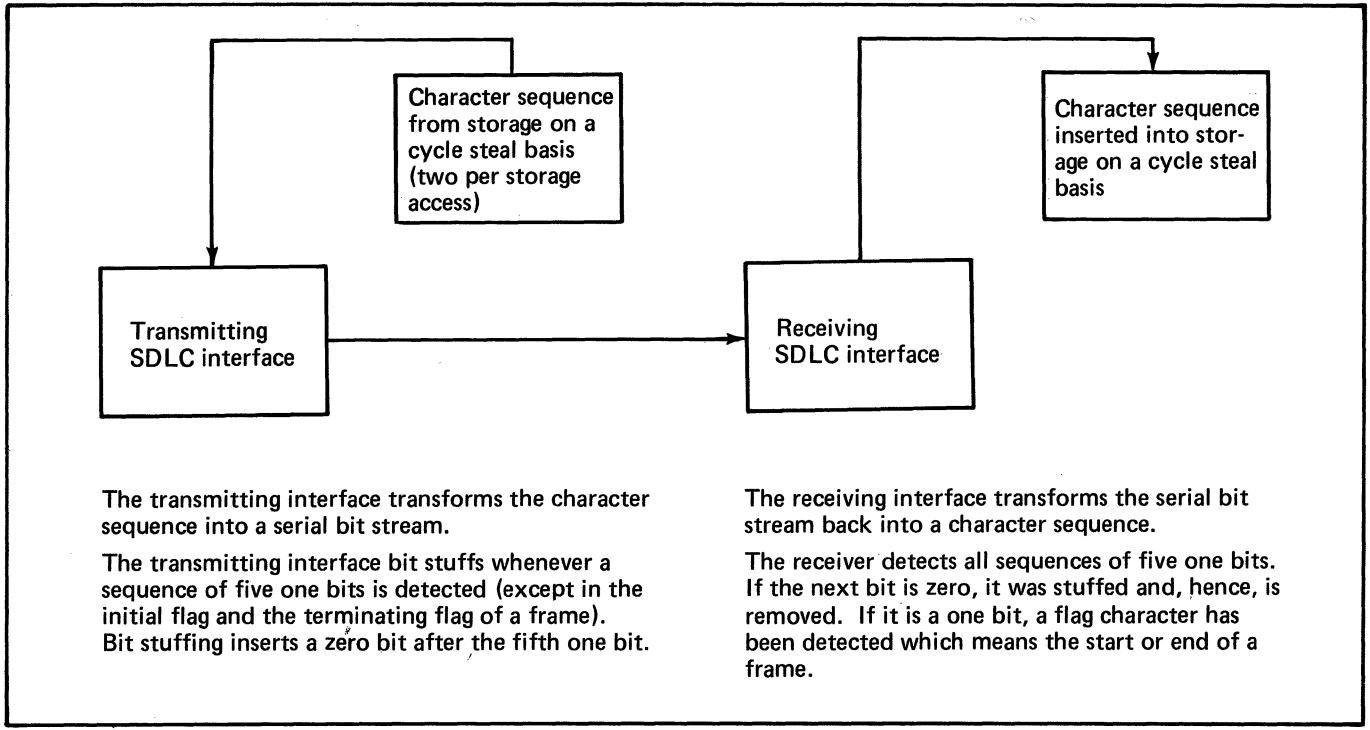


Figure 125. Bit stuffing (1 of 2)

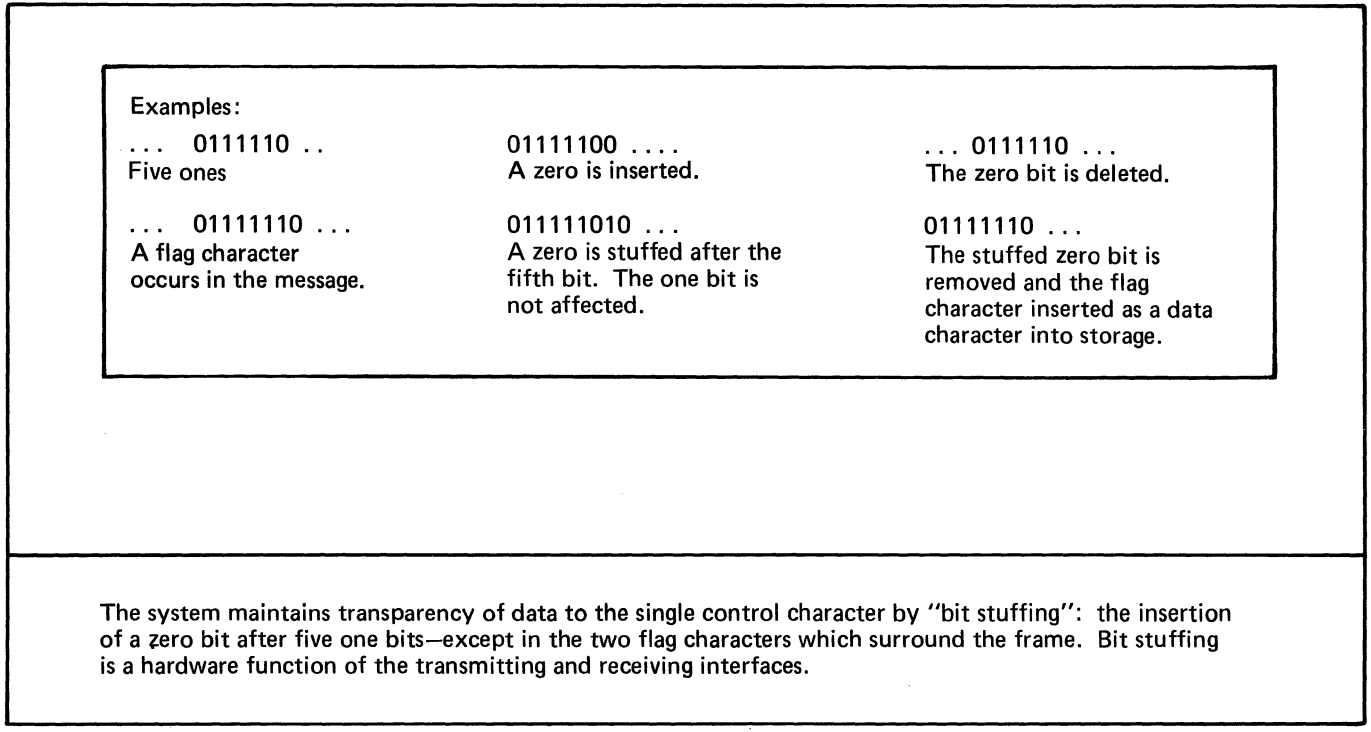


Figure 125. Bit stuffing (2 of 2)

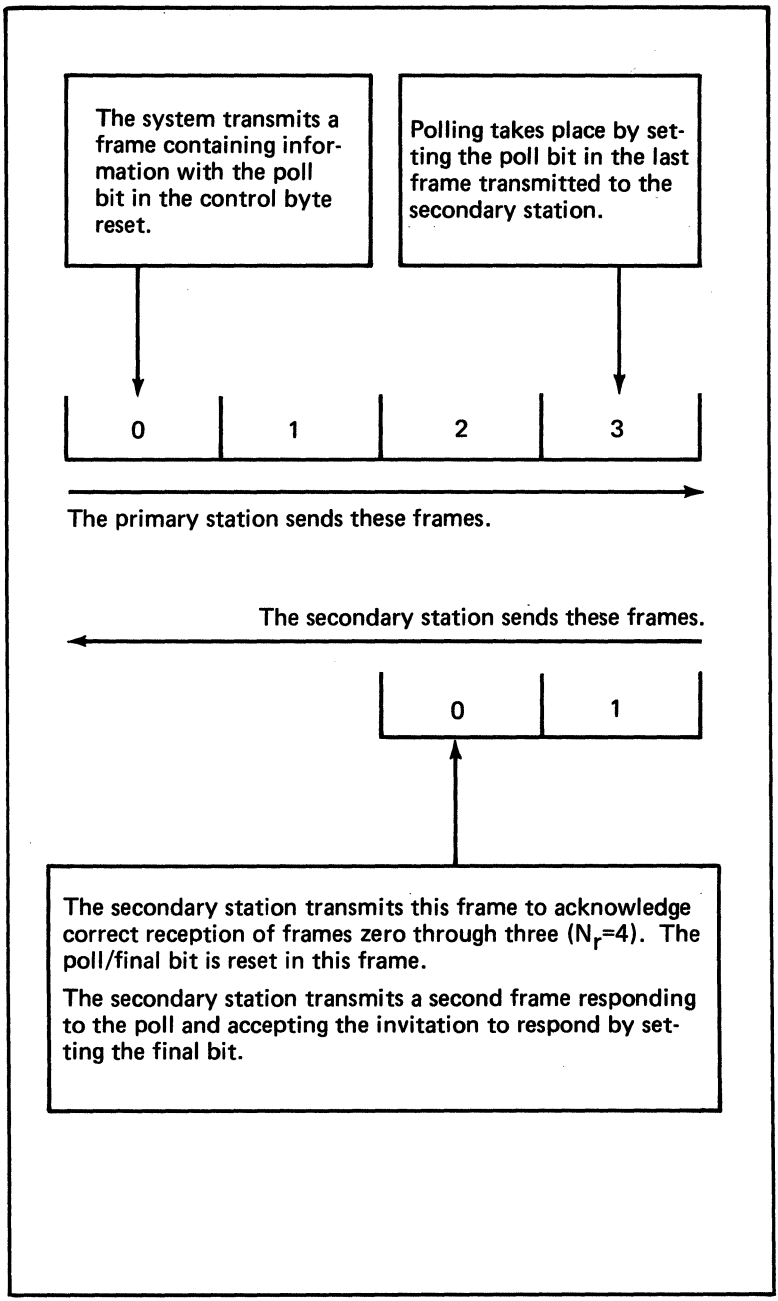


Figure 126. Polling takes place with the single P/F bit within the control byte of a frame

important because they are used for data link management including:

- Activating and initializing secondary stations
- Controlling response mode of stations
- Handling procedural errors which cannot be resolved by retransmission

SDLC Interfaces

The Series/1 systems support SDLC with one interface which handles one half-duplex line at bit rates up to 9,600 bits per second. The system supports communications in the full-duplex mode via two of these interfaces: one to handle each direction. In addition to handling the general SDLC protocol as described, IBM has built the interface to further interpret fields in standard ways so that a variety of conventional terminals can be used directly with this communications' protocol in exactly the same way they are used with the IBM System/370. For other terminal communications, it is the responsibility of the processor and the receiving interface to interpret information fields in an agreed upon manner. For processor to processor communications, SDLC provides a mechanism for full-duplex transmission of varying length messages in a particularly efficient manner.

The Series/1 SDLC interface operates in one of three modes: monitor, receive, or transmit. In the monitor mode, the interface monitors the line for a flag character. If the interface is a primary station, it immediately enters the receive mode. If the interface is a secondary station, it checks the address following the flag to see if it is the station addressed. If so, it enters the receive mode; if not, it remains in the monitor mode. In the receive mode, the system corrects the incoming bit stream to a byte sequence and enters it into storage. The entry begins with either the address field or the control field depending upon whether the station is acting in a primary or secondary role. A primary station needs the address byte to check that the message is from the proper secondary station, whereas a secondary station uses that byte only to determine which station has been addressed.

The interface enters transmit mode when commanded to do so by the processor (a cycle steal command with the specific function specified in the device control block). The system then transmits frame-check sequence bytes and a final flag byte (not stuffed) to complete the frame. If chaining is specified in the device control block, the interface continues with the next frame.

Integration of Communications' Support Software into the Series/1

A hardware interface—which uses the cycle steal channel to access character sequences in main storage for transmission, and to input character sequences into main storage—supports each communications' protocol described (asynchronous, binary synchronous, and SDLC). An application communicating with a remote device must then set up the character sequences it wishes to transmit and, by means of cycle steal input/output commands, pass them to the interfaces. Similarly, interrupts to the processor terminate messages inserted into main storage so that an application program can interpret and use the information.

Dedicated applications often use communications' support in just this way because it provides all features and capabilities in a simple fashion, and the user can tailor the dedicated programs exactly to fit the application. This is especially important in applications where the primary purpose of the processor is to support communications. However, many applications use the communications' system simply to handle devices. For these applications, the Realtime Programming System provides software support of communications' functions using asynchronous and binary synchronous protocols and SDLC. With these protocols, the user can, in effect, treat remote devices or processors in the same way as any other device connected directly to the processor.

Communications' Software Organization

Software use of communications is best organized as shown in Figure 127 with an application task: 1) processing

data and communicating it to remote processors or devices through the communications' task, or 2) processing data received from remote processors or devices by way of the communications' task. Separating the processing and the communications' task simplifies creation, debugging, modification, and documentation of these applications. The Realtime Programming System makes communications among tasks very straightforward so that it is feasible to define simple intertask interfaces which allow modification of either the communications' task alone or the processing task alone.

Because intertask communications are strongly supported at all levels in the Realtime Programming System, the processing part of the application task can be written in assembly language, FORTRAN, or PL/I as appropriate for the application. Notice that PL/I is highly effective for such applications because of its extensive character and string manipulation capability. This capability is one reason that a modern language like PL/I is so appropriate for online small computer applications, and is another good example of the *integrated* design of the Series/1 processor, interface hardware, and software support.

One important Realtime Programming System feature is its support of the Series/1 as a cluster controller in an SNA (system network architecture) IBM System/370 network configuration. Essentially, this means that the Series/1 small computer can be used in multi-computer applications similar to those for which special purpose devices have been designed in the past. In particular, the burden of software support of the network is provided under the operating system so that users can concentrate on their applications. This fact is very important to OEM users who are building application systems which must be compatible with host computer systems.

The communications' portion of the user-written tasks involves assembly language statements which define characteristics of the devices and the details of communications. The system provides a series of macros to simplify this task.

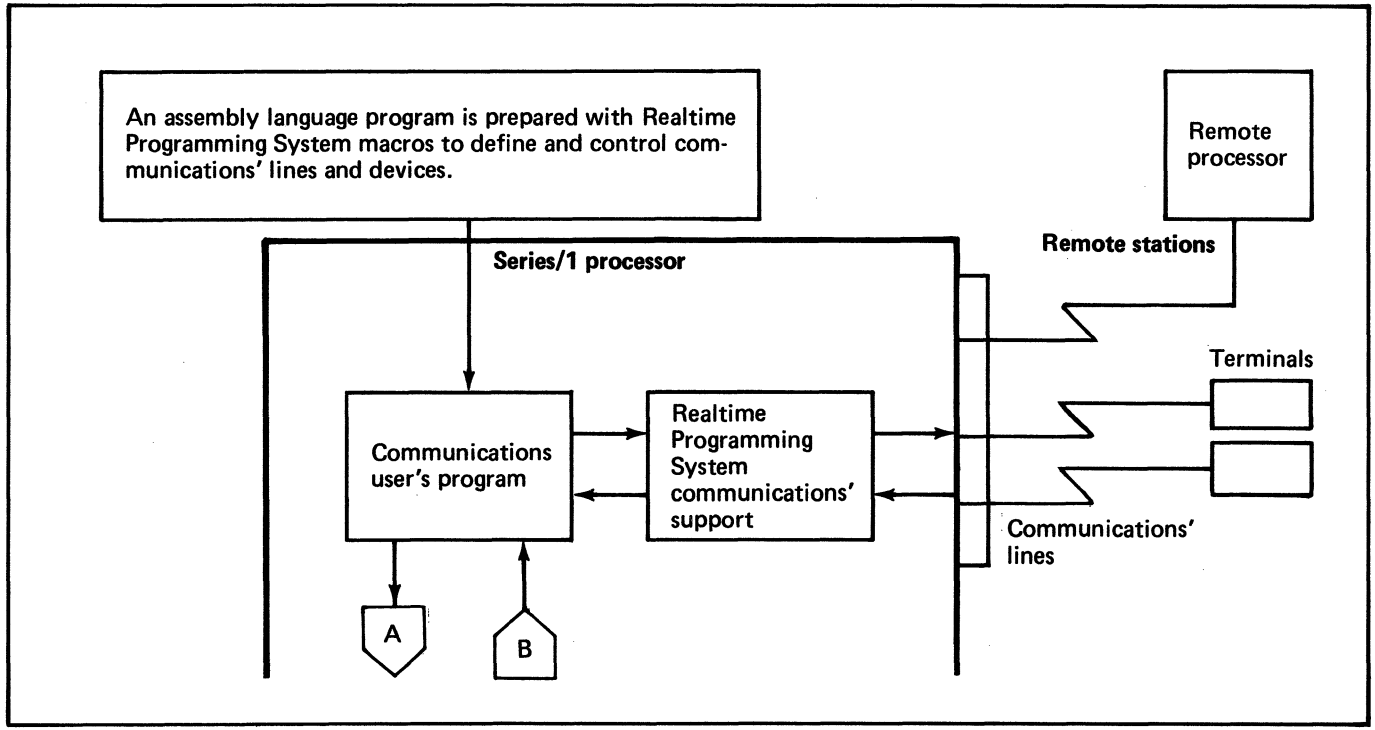
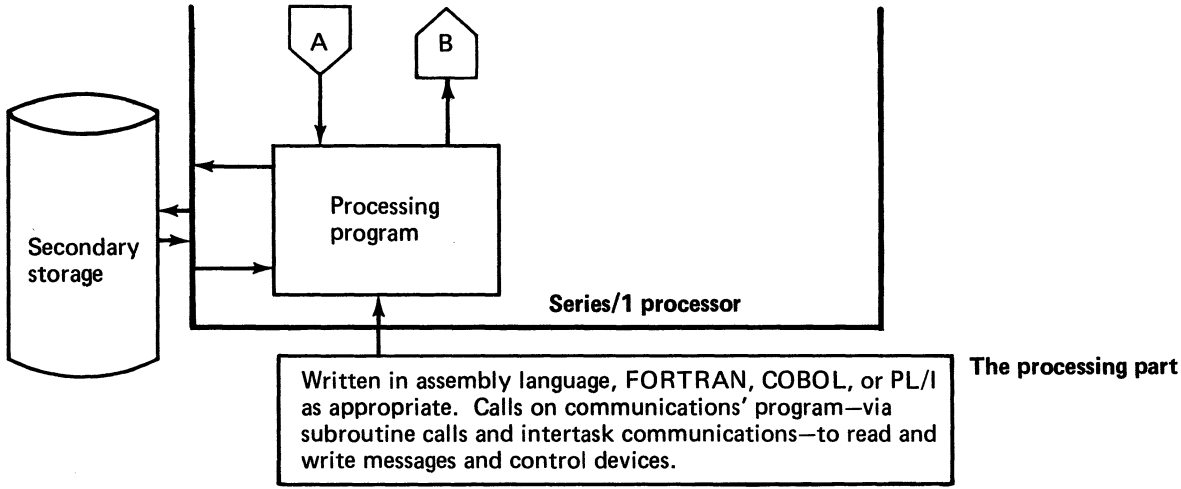


Figure 127. Software use of communications (1 of 2)



The organization of communications' software usually separates the application processing program from the user's communications' program; this organization enables the appropriate level of programming language to be used for the two programs' implementation.

Figure 127. Software use of communications (2 of 2)

These macros support:

- Definition characteristics of the remote stations and associated communications' lines
- Transmitting and receiving data
- Breaking of connections
- Establishing a list of remote station identifiers for communications with dial-up or switched facilities

It is not appropriate here to describe in detail the macros which specify device characteristics. It should be noted that these macros exist and that the system can apply all the characteristics of any data set in a local file to remote devices or processors using either the higher-level language or macro assembly language statements like Connect, Disconnect, Read/Write, Open, and Close.

Event-Driven Software

Communications with a remote processor or terminal are not instantaneous. In fact, if the communications' line is noisy, multiple transmissions might occur automatically at the first level of protocol. Consequently, it is not practical for an application task to depend upon precise timing. This condition occurs in all realtime, online applications and realtime programming techniques handle it: scheduling on the basis of time, internal events, and external events or interrupts. The Series/1 software architecture includes these mechanisms to enable application tasks:

- Signal the occurrence of an internal event (post the event)
- Schedule a task to become active when an event occurs (wait for event)
- Be organized to respond to an external interrupt

The same event and interrupt mechanism is used throughout the Realtime Programming System and is also available for communications' software. An event is associated with each message transmitted or requested. The user can designate tasks to operate in a variety of ways:

- Wait to receive the message (that is, become inactive but

be reinitiated by the operating system when the message arrives)

- Not wait for the message (as in double buffering of input)
- Operate in any other manner that the program designer chooses

The important consideration is that the system uses the same techniques to solve realtime synchronization problems when communications are involved as it uses when those problems arise elsewhere in the data processing environment. Because event and interrupt mechanisms are integrated into the overall hardware and software architecture, the system can solve these problems expeditiously.

Dedicated Hardware and Software Support for Communications: The Programmable Communications Subsystem

IBM has carefully integrated support of communications-based applications into the Series/1 hardware and software architecture as indicated throughout this chapter. Even at the indicated level of support, handling large numbers of terminals is difficult because they require considerable custom software support and a major portion of the Series/1 processor's capability. When a communications-oriented application reaches a certain size, the user must off-load the processor, performing some or many of the required functions in dedicated hardware or separate processors. At the same time, however, this hardware should retain the integrated architectural features of the Series/1.

Subsystem Architecture

To handle applications involving large numbers of terminals and communications' lines, IBM has provided the Programmable Communications Subsystem as part of the overall Series/1 architecture. This subsystem is essentially a separate processor which handles many of the functions required to support a variety of lines, line speeds, terminal

types, and protocols. At the same time, the subsystem is integrated into the Series/1 hardware and software architecture so that the same measure of self-diagnosis, availability, and error recoverability can be achieved with the subsystem as with the parent system itself. Furthermore, the subsystem is fully compatible with the Series/1 software architecture: Realtime Programming System support and Program Preparation System support are complete.

Figure 128 lists some of the communications' functions which require detailed software design to effectively serve large communications-oriented applications. Notice in particular that many of these problem areas are unique to communications-oriented applications: control of modems, telephone call answering and originating, and redundancy checking. Although conceptually simple—and able to be implemented completely within the main processor—these tasks can incur significant overhead when large numbers of different communications' lines and terminals are involved. The Programmable Communications Subsystem can handle all the areas listed in Figure 128 as well as many others which may be unique to a particular user or application.

Communications' Interfaces

The Programmable Communications Subsystem is a set of standard Series/1 boards which plug into the Series/1 units just like other interfaces and attachments. Figure 129 shows the system in block diagram form. At the lowest level are a variety of interfaces suitable for attachment to a variety of terminals, devices, and telephone lines. Interfaces include:

- Synchronous and asynchronous EIA data set interfaces
- Automatic call handling interface
- Teletype current interface
- Synchronous and asynchronous integrated modems
- SDLC data set and direct interfaces

The common important factors in this list are the variety of line speeds and types of interfaces provided, and the fact that the user may mix all of them in any arbitrary way within

- Buffering
- CRC checking/generation
- LRC checking/generation
- Control character generation/recognition
- Clocking of direct connect hookups
- Data chaining to/from storage
- Auto-answer
- Break function
- Case shift
- Timeouts or interval timer
- Modem control
- Auto-poll
- Auto-call control sequencing
- Console control
- Internal self-diagnosing RAS features
- Trace

All of these functions are characteristic of communications' applications and the system can provide them directly within the Series/1 application software. When the number of terminals and the variety of terminals and lines is large, it is more efficient to off-load standard communications' functions into a special processor called the Programmable Communications Subsystem.

Figure 128. Basic functions provided by the Programmable Communications Subsystem

the communications' subsystem. Notice that many of these interfaces provide facilities similar to those available for direct interfacing to the Series/1. Of course, a different level of support is available through the communications' subsystem.

The system provides support of these interfaces through the scanner portion of the subsystem as shown in Figure 129. This hardware scans the interfaces at speeds ranging from 45 to 1200 bits/second using an internal clock, and up to 9600 bits/second using the clock in a data set attachment. These

figures correspond to the scanning rate of each interface so that the combination rate is very much higher. In addition to scanning the interfaces to collect or transmit characters (deserialize the input or serialize the output), the scanner provides many of the capabilities discussed earlier for individual communications' interfaces:

- Programmable synchronization and line turnaround characters
- Programmable selection of bits per character
- Parity checking
- Programmable selection of the number of stop bits for asynchronous terminals

Thus, in the single scanner hardware, the communications' subsystem provides the same communications' support that is built into the separate communications' interfaces previously discussed. In this way, the Series/1 provides communications' support for large numbers of terminals at low cost.

Subsystem Controller

The heart of the communications' subsystem is the controller which contains a processor, read-only storage, and a writable storage called control storage. It is this controller which the end user or OEM user can program to customize the communications' subsystem by handling a particular group of terminals and a particular application. Figure 130 shows the basic organization of the controller.

The line control block is a user-defined data area; the system provides one block for each communications' line attached to the subsystem. It contains parameters describing the line, address of buffer areas, status information, and similar information necessary to handle transmission of information on the line. In addition, the line control block contains a pointer (address) for another block of data—the function address table. Essentially, this table is a list of subroutines or program segments—called function strings—which perform the individual operations appropriate to that particular line. For example, function strings might be provided to support a binary synchronous protocol or a special

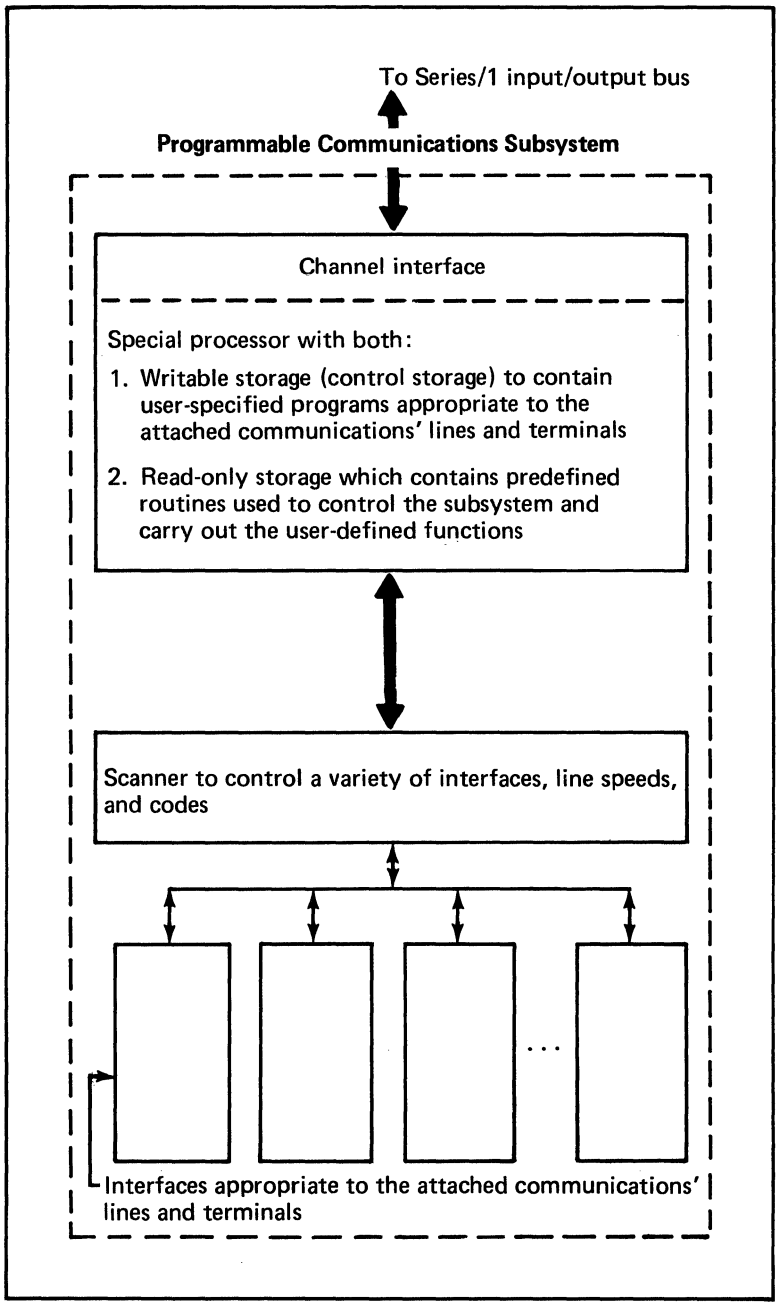


Figure 129. Hardware organization of the Programmable Communications Subsystem

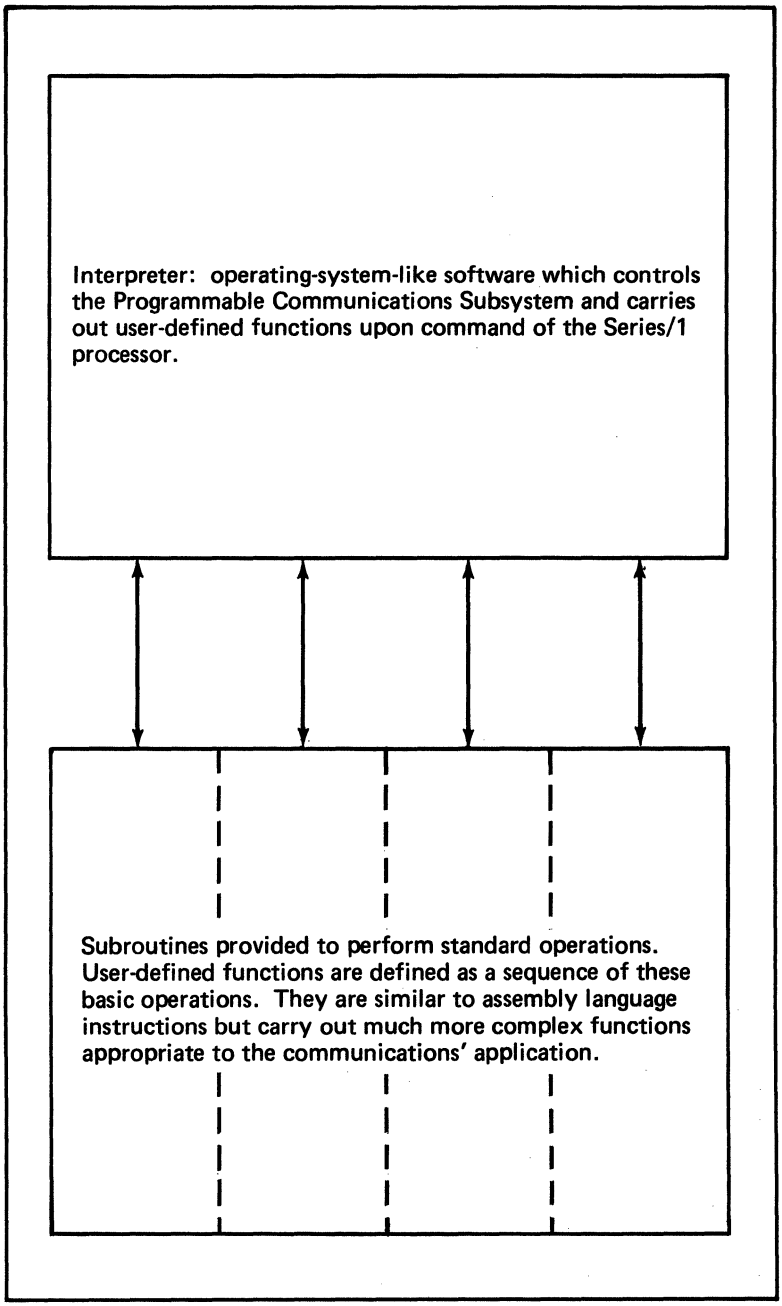


Figure 130. Software organization within the Programmable Communications Subsystem (1 of 2)

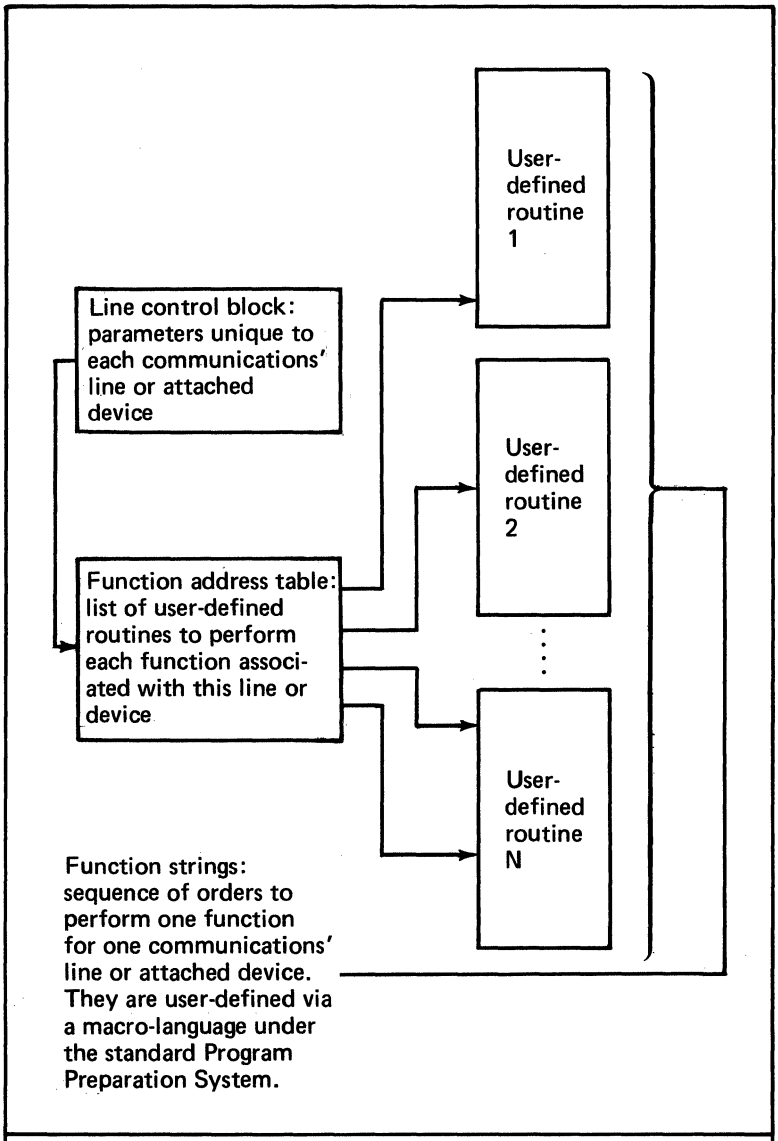


Figure 130. Software organization within the Programmable Communications Subsystem (2 of 2)

purpose protocol, or function strings might be used to error-recover when certain conditions are detected. In effect, the function address table associated with each line is a list of code segments that handle each situation which arises when carrying out communications across that line.

Line Control Software

The user must define each function to be carried out for each line, and produce the subroutines or program segments (function strings). The Programmable Communications Subsystem facilitates preparation of these function strings by providing the interpreter shown in Figure 130. The interpreter can handle approximately 90 pre-defined operations which function like instruction operation codes in a computer. Operations include:

- Transmit or receive data
- Block check character control
- Automatic polling
- Control of modems
- Timer control
- Branch and Link instruction

It is not appropriate to list all operations in detail here. The reader should consult the appropriate Programmable Communications Subsystem programming manuals for this information. It is important to indicate here that users prepare their programs or function strings in a sequence like the type of high-level instructions listed above. This sequencing facilitates the construction of rather elaborate communications' support programs in the subsystem. Source code format function strings are also available for terminals like the 3270 family, the 2740, and Teletype Models 33 or 35.

User-Generated Software

The programs within the Programmable Communications Subsystem are microprogrammed. Instead of requiring the user to learn programming techniques at this detailed level, IBM provides an elaborate programming support system

which runs under the normal Program Preparation System software. This support system means that the user can create function strings and necessary tables simply by calling macros. The user enjoys a dual advantage: program preparation is at a high level, while microprogrammed subsystems provide the user with all the advantages of an efficient data processing operation.

It is important that the hardware and software architecture of the small computer integrate any subsystem like the communications' subsystem. With the Series/1, hardware integration is simplified because the architecture of the input/output system permits processors to be added to the system without sacrificing the communications' rates to the main processor and main storage. A previously illustrated example of these subsystems was the floating-point subsystem which has both an objective and an architectural structure similar to that of the communications' subsystem.

Integrated Software Structure

Figure 131 shows the integration of the software support for the communications' subsystem. As previously indicated, good programming practice separates communications' software support into different modules to support different terminals and different application requirements. The communications' subsystem support extends this architecture as shown in the illustration. The one module has been completely off-loaded into the communications' subsystem as discussed above. A simple interface remains which drives the Programmable Communications Subsystem using normal input/output instructions. As shown in the figure, this interface permits an application program to write Execute I/O instructions in which the immediate device control block references a device control block—as in all cycle steal input/output operations. Within the device control block, the system codes a command which is effectively an index into a function address table within the Programmable Communications Subsystem. The subsystem, in turn, is a pointer to a function string which performs the operation. In other words, the user supplied I/O instruction is equivalent

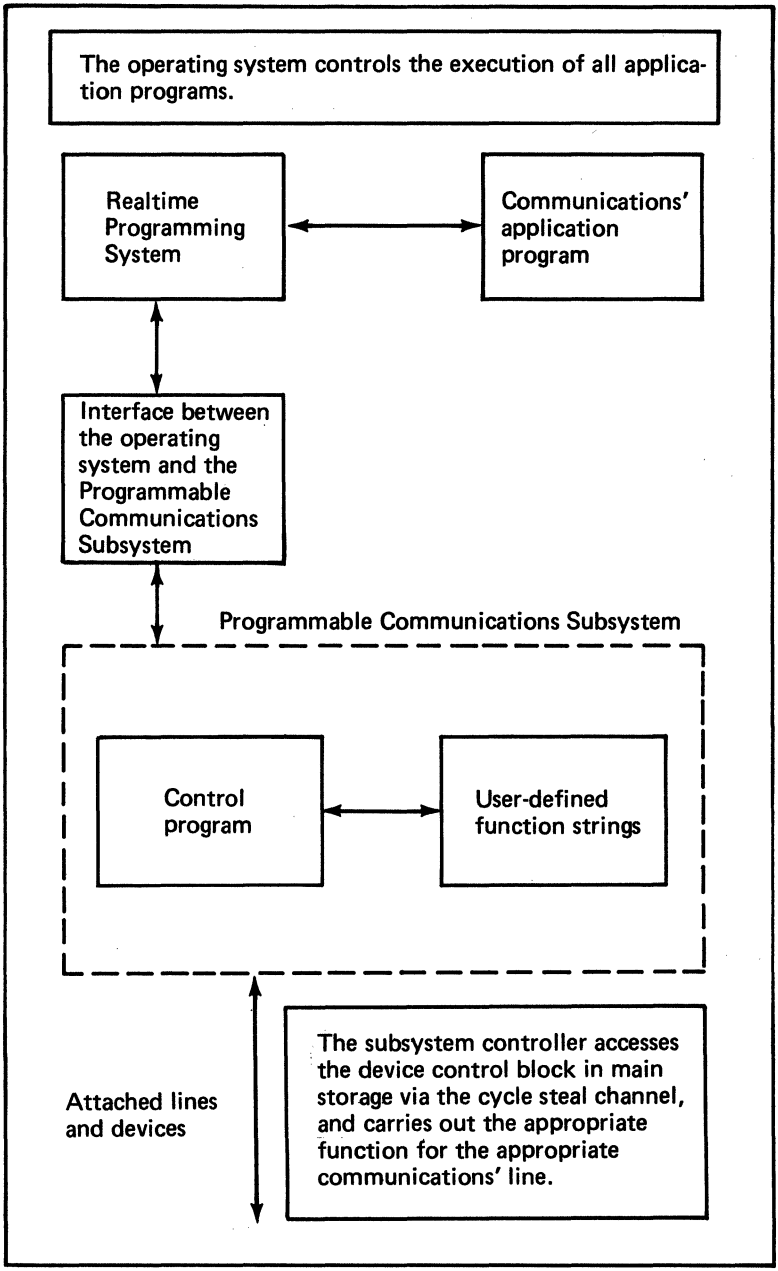


Figure 131. Integrating software support of the Programmable Communications Subsystem into the Realtime Programming System operating system (1 of 2)

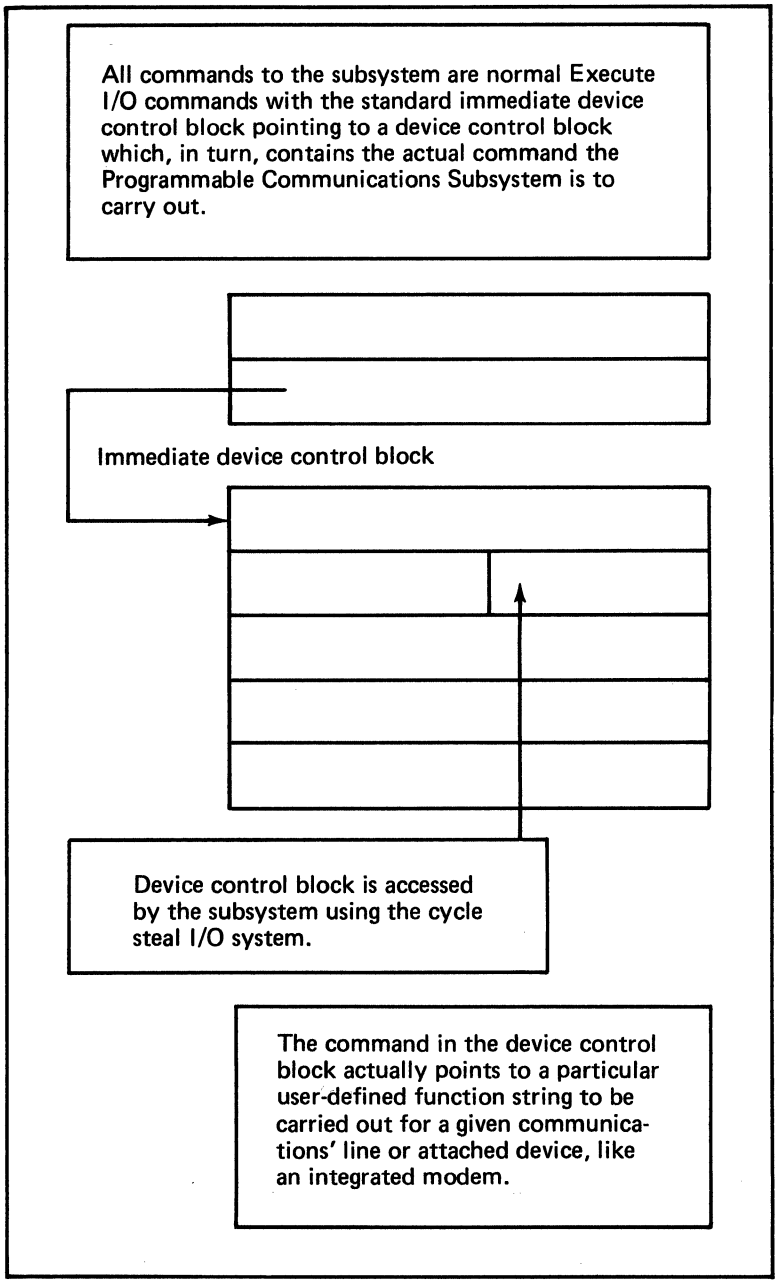
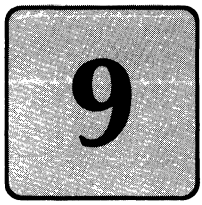


Figure 131. Integrating software support of the Programmable Communications Subsystem into the Realtime Programming System operating system (2 of 2)

to a call to one of the user-defined function strings or subroutines within the Programmable Communications Subsystem. Conceptually, it does not matter whether such communications' subroutines are in the main processor or in a separate subsystem. The user maintains complete control over what function strings the system provides and what they do. In this manner, users may easily customize their communications' systems. Because of the interface within the Series/1 Realtime Programming System software, the user can construct application programs either in a higher-level language or in assembly language to perform two separate functions: 1) to use information gathered through the communications' system; 2) to generate information to be transmitted through the system.

The combination of hardware and software communications' architecture in the Series/1, then, provides support to customize applications at similar levels whether they involve a small or large number of terminals. The Series/1 achieves this support while combining availability, reliability, software support, and compatibility with other systems—all requirements listed in Chapter 1 for successful, communications-oriented small computer applications.



Reliability, Availability, and Serviceability (RAS)

The success of any small computer application depends upon the close cooperation of the hardware, software, and maintenance systems. As stressed in earlier chapters of this book, if these three components are not fully integrated, the overall system will be less successful. Hardware and software integration have been discussed earlier, with emphasis on:

- How hardware is present to support the appropriate software for small computer applications
- How software takes advantage of the hardware
  - To carry out the application efficiently
  - To minimize development and debugging time

In the same way, hardware and software must be designed so that the resulting system will be reliable and maintainable. The objective of this chapter is to discuss how maintenance is integrated into the overall IBM Series/1 hardware/software design.

The Contribution of Maintainability to the Overall System

The combination of hardware and software designed to carry out an application often includes both IBM- and third

party-supplied hardware and software components; the combination must operate reliably in a realworld environment. It would be unrealistic to expect such a system to operate without problems or failures. Furthermore, it would be prohibitively expensive to design every system so that the probability of failure would be almost non-existent—for instance, as low as the failure ratio of manned space flights in recent years. The design objective of the Series/1 was to devise reliable hardware and software while simultaneously providing the system with a quick and efficient problem response capability to minimize the *effects* of failures. Figure 132 shows the various states of a small computer system. In normal operation, the system performs its intended function. A "soft" error condition—an error which the system can detect or bypass without halting operation—must be identified and responded to rapidly. To do this, the hardware and system design must function in a manner to detect these errors accurately and easily.

An example of such an error could be the transmission of a noise-corrupted byte of data between a device and storage—an error which the system might correct by a retransmission. The system must detect these soft errors by using:

- Parity bits on the data
- Error detection bits, where appropriate
- Checks sums or cyclic redundancy codes
- Echoing of data
- Other procedures depending upon the devices and distances involved, and the criticality of the data

The combination of hardware error detection and software error recovery procedures increases the reliability of the system. When a "hard" error occurs—an error which is severe enough to halt operation of the system—time is required to diagnose the source of error and perform the necessary maintenance. If the system can reduce the time required to diagnose and repair the error, then the effect on the application can be minimized.

Availability of the system is the net time that the system is actually available to perform the application; it is, to the

user, the most important measurable element in the system. A high level of availability implies a minimal response time to hard errors and rapid recovery from soft errors.

It is important to note that every component of the system—hardware and software—must be integrated into the soft and hard error detection and recovery procedures. This integration is essential because it is typical of small computer applications that a variety of OEM- and vendor-supplied hardware and software components interact closely to carry out the application. It is important to emphasize the distinction between the maintenance capability of the Series/1 architecture and the maintenance supplied by IBM itself. IBM has designed the Series/1 so that self-diagnosis and maintenance can be performed effectively and efficiently. OEM users can take advantage of those capabilities in the design and utilization of their devices and interfaces so that the systems they configure can also be maintained effectively and economically. IBM-supplied maintenance covers only devices supplied by IBM.

Design and Organization for Reliability

Design for reliability implies design for low failure probability. There are two aspects of reliability: 1) low failure probability of hardware components, and 2) low probability that either noise corruption or hardware failure will cause the application to malfunction. For example, failure of a component might not cause the system to halt but might cause errors in data stored in files, transmitted to devices, and in other locations. These errors are just as serious as hardware failures. Consider, first, the hardware reliability.

Component and Device Reliability

The Series/1 electronics use extensively the large-scale, integrated TTL logic of the type used and proven in other IBM products. With proper burn-in, testing, and other quality control mechanisms developed in the electronic revolution of the past 15 years, such devices are now sufficiently reliable

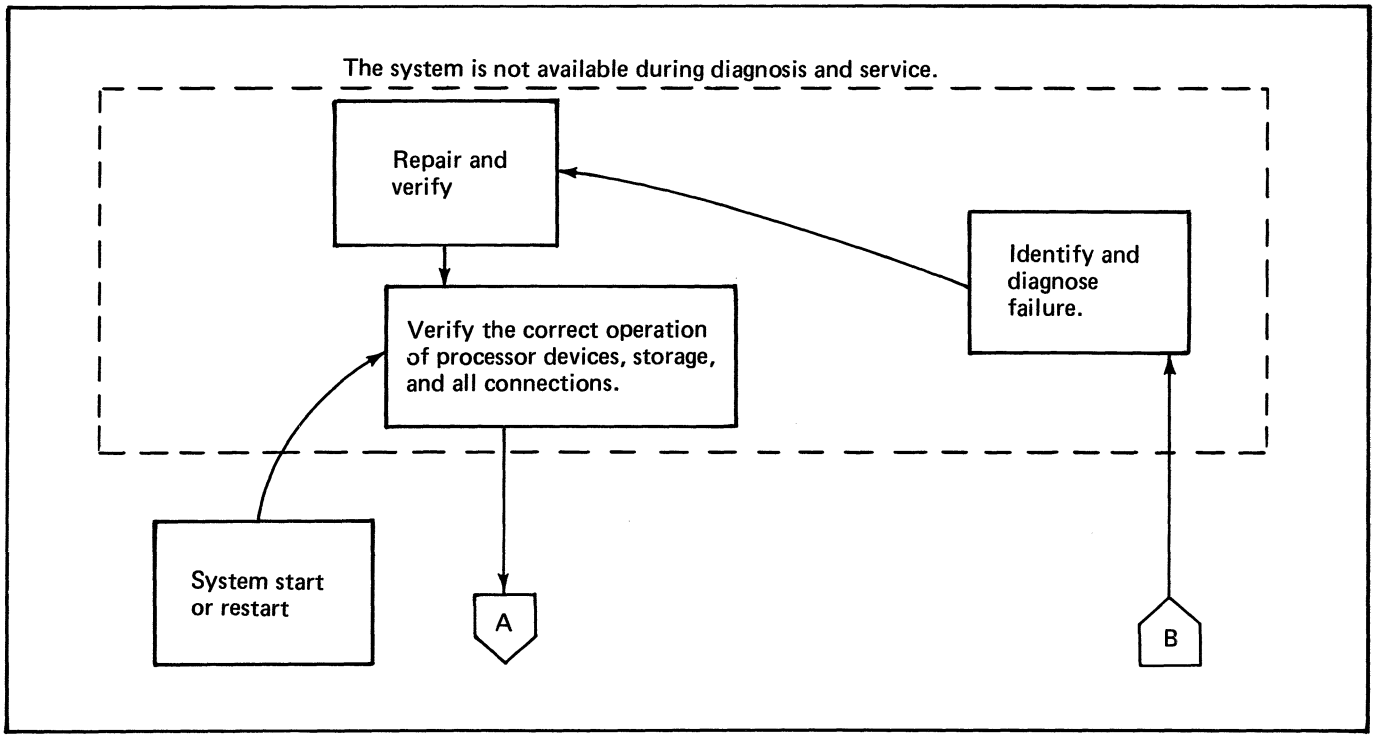


Figure 132. Availability states (1 of 3)

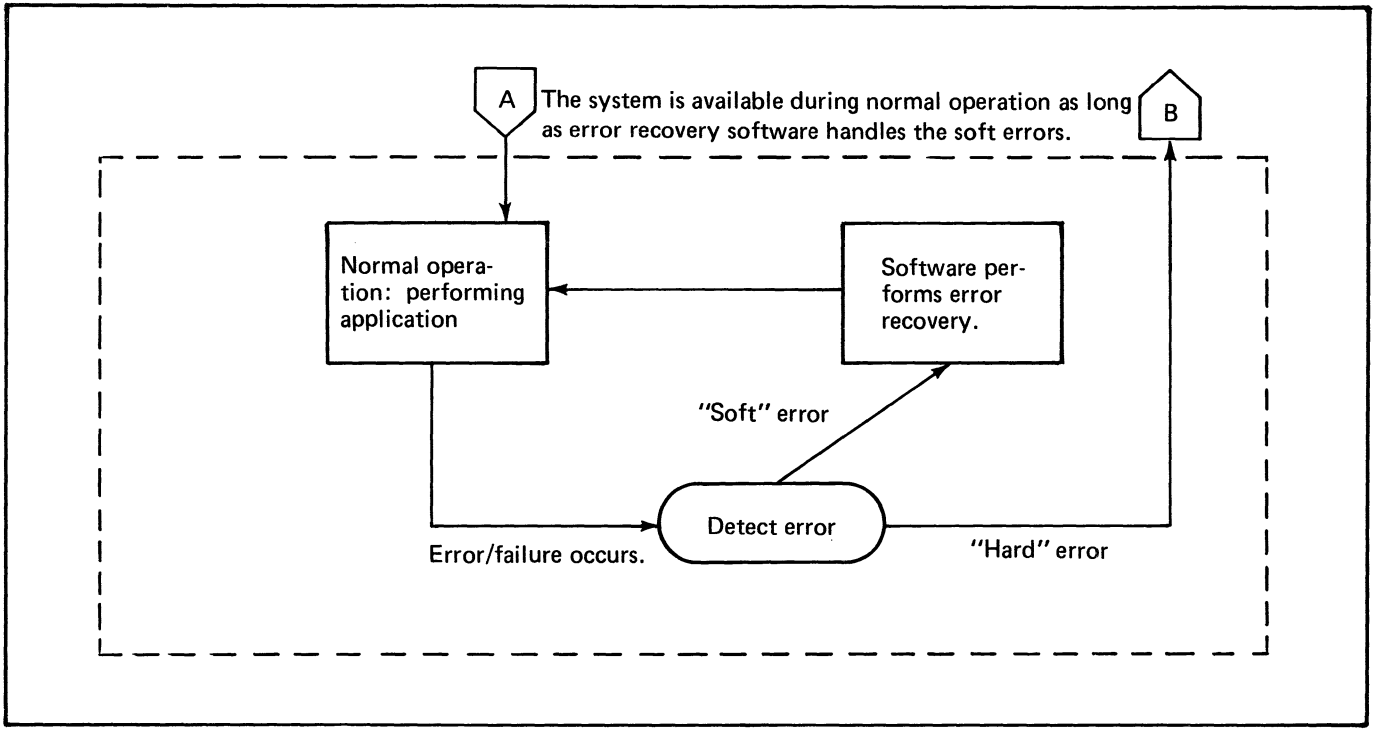


Figure 132. Availability states (2 of 3)

Achievement of high availability requires an integrated system of hardware and software to perform the self-checking and self-diagnosing of errors. The integrated system of hardware and software must be extended by the user to include OEM and application hardware and software.

Availability is achieved in three ways:

1. Reliable hardware minimizes actual failures
2. Extensive self-checking detects soft errors which may be corrected by error recovery procedures built into the system and application software
3. Extensive self-diagnosing identifies and isolates a hard failure quickly to minimize mean-time-to-repair and maximize availability

Figure 132. Availability states (3 of 3)

to meet the availability levels demanded by small computer applications.

Reliability is more important in devices that require mechanical motion which causes wear and vibration. In this area as well, the computer industry has learned to design reliable devices. For example, the IBM disk storage unit uses a sealed disk enclosure containing the fixed disk and the mechanical access mechanism. Sealing the disk eliminates operator handling of critical devices, reduces exposure to external contaminants, and obviates preventive maintenance of heads, disk, and other mechanical devices within the enclosure.

With this design, the probability of the disk's data accuracy and availability is greatly increased. Similarly, slower devices like the Series/1 diskette unit and the line printer use a stepper motor as main drive rather than a continuously running motor assisted by a clutch mechanism. Although the latter motor is adequate for the task and perhaps lower in initial cost, the stepper motor removes a high-maintenance item (the clutch) from the system. Such design characteristics enable IBM to market devices requiring little preventive maintenance.

Attention to details like these are evident in the specifications of other Series/1 devices. One previously mentioned, important detail is that IBM has specifically designed the Series/1 hardware and software to be responsive to OEM devices. Of course, it is necessary that these devices be as well designed as the Series/1 itself to prevent compromise of overall system reliability.

When hardware is reliable, system availability is extended. Soft errors will continue to occur, however, because of noise on transmission lines, variation in power levels, and human fallibility. Detection of these errors is fundamentally important to, and a major consideration in, the design of the processor and device hardware and software. Error detection is built into the system at all levels as shown in Figure 133.

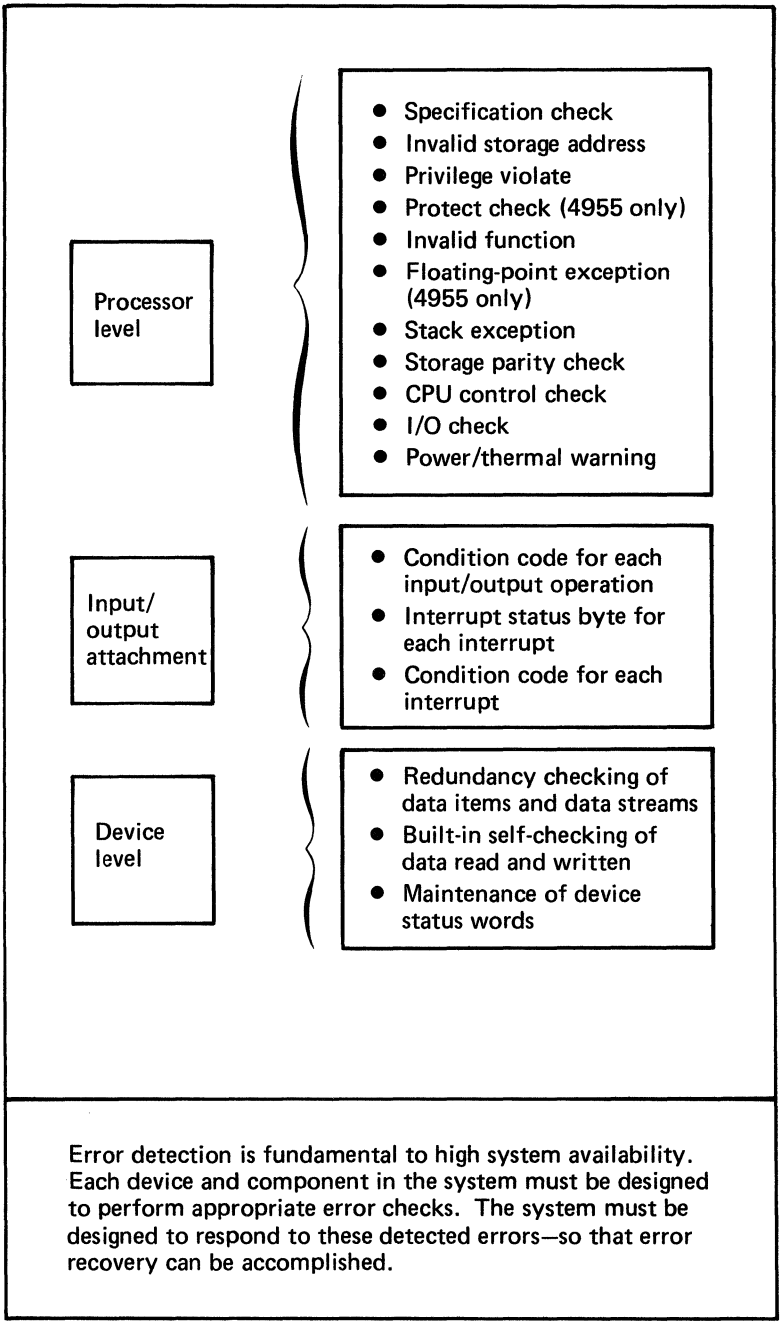


Figure 133. Elements of error detection

Processor Error Detection

At the processor level, some of the built-in error detection techniques (most of which have been previously described) include:

Specification Check. An indirect address or a generated effective address has violated an even-byte boundary requirement.

Invalid Storage Address. One or more words of the instruction or an effective address is outside of the installed storage size of the system.

Privilege Violate. A privileged instruction is encountered while in the problem state.

Protect Check. An instruction is being fetched or data is being accessed from a storage area not assigned to the current operation, or an instruction is attempting to change an operand in a storage area assigned as read-only.

Invalid Function. An illegal operation code or function combination has been detected, or a floating-point operation was attempted and the floating-point feature is not installed.

Floating-Point Exception. An exception condition is detected by the optional, floating-point processor.

Stack Exception. An attempt has been made to pop an operand from an empty main storage stack or push an operand into a full main storage stack; or a stack cannot contain the number of words to be stored by a Store Multiple instruction.

Storage Parity Check. A parity error has been detected while data is being read out of storage by the processor.

CPU Control Check. The hardware has detected a malfunction of the processor controls (e.g., no level is active but execution is continuing).

I/O Check. Hardware error has occurred on an input/output interface.

Power/Thermal Warning. A power failure or thermal overload has occurred.

Note that not all of the conditions listed above pertain to all processor models.

Detection of any of these conditions permits an increase in system reliability because error recovery operations can be initiated (Figure 134), thereby preventing the system from actually halting. Software design must recognize the hardware assistance in recovery operations and incorporate this assistance within its own performance. Consequently, software design is crucial here. An example of recovery software design and performance is illustrated in Figure 135 where the response to a power failure is depicted.

Battery Backup

If the source voltage drops below approximately 85 percent of the normal line voltage and the system includes a battery backup unit configuration, the system will automatically switch to battery power and will continue to power the processor. A class interrupt occurs causing a branch to the power/thermal interrupt handler routine which can continue to monitor the power/thermal failure bit in the processor status word. After mainline voltage is restored:

- The system will automatically switch back to mainline power
- The power/thermal failure bit in the program status word will be turned off
- The system can resume execution of the problem program

No data will be lost from main storage. Equally important is the fact that the system will not generate erroneous results because of low voltage levels in storage or on the input/output channels. Thus, the system protects the application from data distortion as well as data loss.

If the configuration does not include a battery backup unit, the system will power down. Upon restoration of the mainline voltage, the system may automatically power itself back up and can automatically re-IPL. The user can program the IPL bootstrap program to reload any program, and resume execution. This automatic restart feature makes the system particularly viable for use in a remote or unattended location.

Input/Output Error Detection

The second level of error detection in Figure 133 is on the input/output channel itself. IBM has designed the system so that all devices provide the following checks:

- Condition codes—each time the system issues an Operate I/O instruction, the device, controller, or channel immediately reports to the processor a condition code pertaining to execution of the I/O command
- Interrupt status byte (ISB)—if an error condition exists after an I/O operation (for example, a channel parity check), the system presents detailed information on the nature of the error in the ISB
- On devices which present interrupts, the system again presents the condition codes with the I/O interrupt to further define the exact status of the I/O operation

With these checks built into each input/output or interrupt operation, the user can create software which is not sensitive to those infrequent errors which do occur.

Device Error Detection

At the device level in Figure 133, each device itself is responsible for checking its own operation; each device signals its errors using the condition code and interrupt facilities of the system. For example: both the disk and diskette units generate cyclic redundancy check characters for both the sector identification field and the sector data field within each sector. Asynchronous communications' interfaces provide longitudinal and vertical redundancy checking. Binary

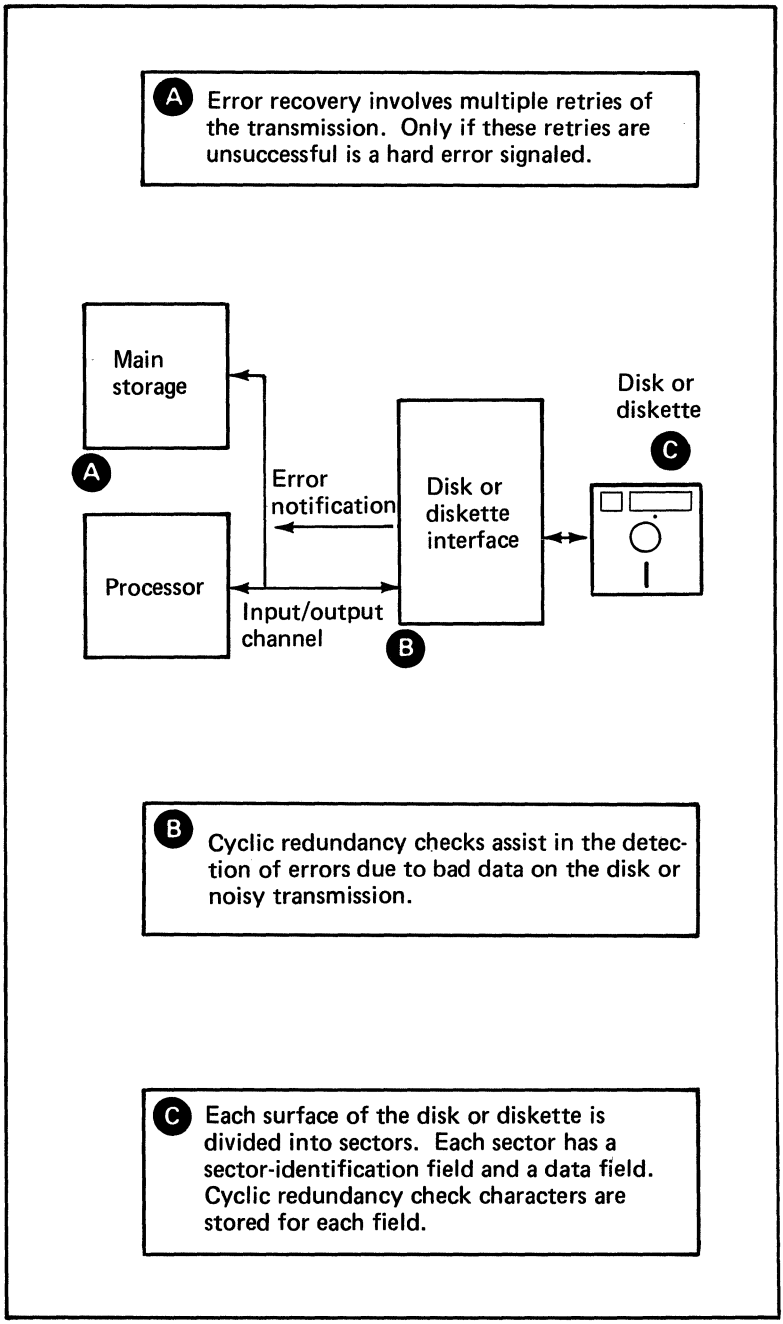


Figure 134. Disk and diskette error detection (1 of 2)

A sector identification field insures that data cannot be read from or written into the wrong area—such action would constitute a major error if it occurred.

Both system software and user-written software for dedicated systems must provide for the error detection capabilities, and both must have a built-in error recovery capability.

Disk and diskette storage error detection illustrates how error checking must be included in the hardware and software system design in order to detect errors appropriately.

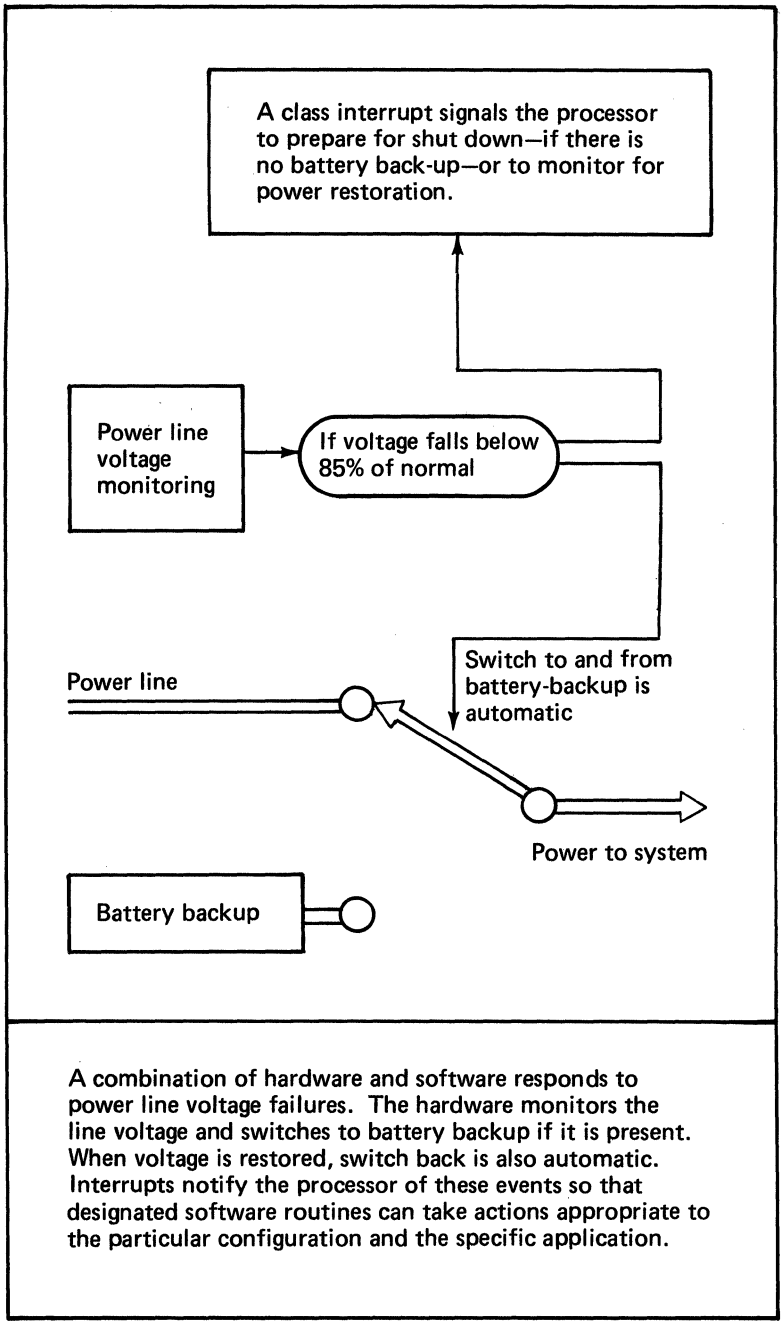
Figure 134. Disk and diskette error detection (2 of 2)

synchronous communications' interfaces offer cyclic redundancy checking. Other devices provide similar appropriate checking for the device. The result of this checking is as follows:

- The integrated design of the system identifies detectable errors
- The system notifies the processor of the error
- If possible, the software will respond to correct the error

Frequently, retry of a storage read or other operation will correct the error. Even if the error is corrected automatically, system software maintains a log of the error occurrence for later system diagnosis by IBM customer engineering or by the user.

In the past, manufacturers have found it excessively expensive to build such extensive self-checking into the hardware. The construction of the microprocessor has reduced these costs to the point where this hardware function is now economically feasible. The effective use of the powerful microprocessor technology to incorporate this self-checking hardware capability has made a highly significant contribution to overall system availability.



A combination of hardware and software responds to power line voltage failures. The hardware monitors the line voltage and switches to battery backup if it is present. When voltage is restored, switch back is also automatic. Interrupts notify the processor of these events so that designated software routines can take actions appropriate to the particular configuration and the specific application.

Figure 135. Hardware and software response to a power failure

Error Diagnosis: The Key to High Availability

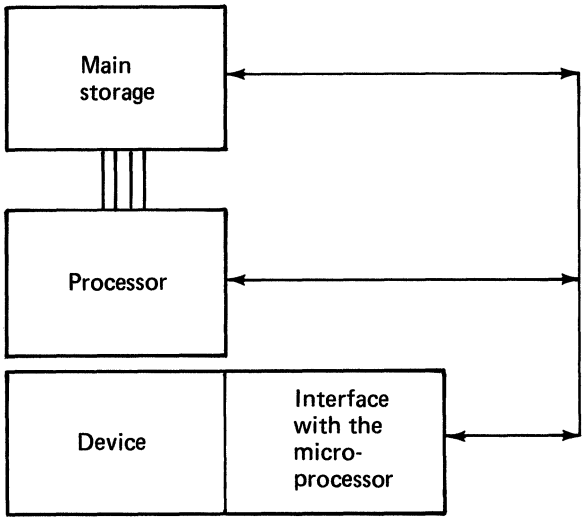
If the system is to maintain its availability at a high level, it must diagnose, quickly, any hard error or failure in any component of the system. Using microprocessors in interfaces and microprogramming in the processor become economically significant factors when the system must perform a high level of error diagnosis. This is so because a system with distributed intelligence capability can contain many more extensive self-diagnostics more economically than can a system with fixed logic designs. Series/1 devices and interfaces use such approaches where it is most economical and appropriate.

Microprocessor Based Self-Diagnosis

The system includes microverification routines in all microprocessors and in the microprogrammed processor itself. These routines are executed when the system is powered-on, reset, or initial program loaded. As shown in Figure 136, the processor is self-checked to assure correct operation—including data flow to and from registers—of the microprogrammed system. Self-checking involves:

- Moving data into and out of registers and checking for an expected result
- Performing micro instructions and comparing the results to known values or checksums
- Other similar procedures

In parallel with the processor's self-checking, the system logically isolates each device controller. Each controller performs a similar self-check including a run through all of its micro instructions and a compare of an accumulated checksum with a preprogrammed checksum. The system then checks by writing and reading back a specified bit pattern in the first 16 kilobytes of storage. Finally, if all modules pass these self-diagnostics, the system is integrated and the input/output channel itself is checked to insure that data can be passed back and forth between main storage and the



The system:

- Writes prespecified patterns into the first 16K-bytes of main storage, and then reads them back
- Writes into processor registers, and then reads them back
- Exercises micro instruction data paths and verifies the results
- Verifies transmission between devices and the main storage across the input/output channel
- Checks the device in a device-dependent manner through its microprocessor-controlled interface

The microprocessor checks for correct control store contents, and verifies data paths.

The system uses the microprogrammed processor and the microprocessor-based interfaces in such a manner that, before the start of the application software:

- The separate operation of each component can be checked
- The combination of devices can be checked

Figure 136. Processor self-checking

device interfaces. Location zero in main storage is used by the device interfaces for this purpose.

If any of these checks do not succeed, the system has detected a hard failure. Because of the detailed nature of the checking, the failure has probably been isolated to a specific printed circuit card. Replacement of the card can usually remove the problem.

If all tests are successful, system operation begins. Detection of a hard failure halts the system; at that point, the diagnostic capabilities of the system are again activated.

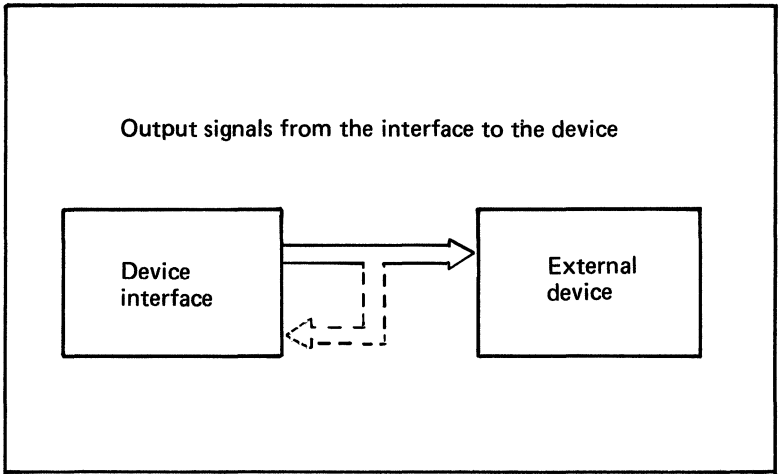
Diagnostic Software

Series/1 provides diagnostic software which takes advantage of the integrated design of the system's hardware and software. For example, the system provides a diagnostic instruction which enables software to check operation of devices at very detailed hardware levels. Storage can be checked by loading specific addresses and data into storage address and data registers, and then checking the result of the storage read or write operation. Similarly, software can check those diagnostic device commands which produce device-dependent results.

Devices are specifically designed to be diagnosed this way. For example, Series/1 provides diagnostic wrap-back facilities for the teletypewriter:

- Attachment
- Timer
- Communications
- Integrated digital input/output
- Sensor input/output units

Diagnostic wrap-back provides, under program control, routing of output signals back into input ports; as a result, the system can check the complete operation of interfaces by outputting data and reading the same data back in again to insure that it is transmitted correctly (Figure 137). In addition, this wrap-back feature can be used without removing cables or attaching special jumpers—further reducing diagnostic time.



Wrap-back of signals permits the system to read back signals being transmitted to an external device; this insures that the desired signal is the one actually sent.

The system must also diagnose the combination of IBM-supplied interfaces and OEM supplied devices. By wrapping-back signals, operation of each portion of the system can be checked without using cables, or disconnecting the system.

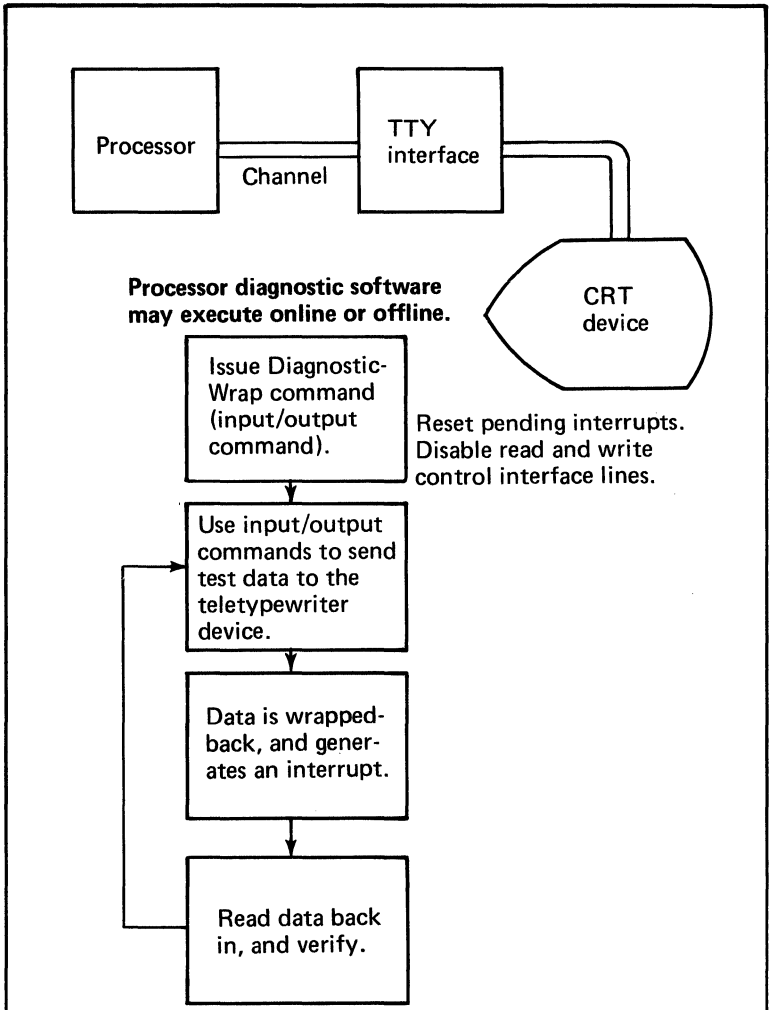
Figure 137. External device diagnosis

Interface Diagnosis

As an example of the power of this diagnostic approach, consider the teletypewriter interface diagnosis, which can take place either with or without a device connected to the interface (Figure 138).

The Reset to Diagnostic Wrap command: 1) resets pending interrupts, condition codes, and all registers in the teletypewriter adapter except the prepare register, and 2) disables the read and write control interface lines. The system places the teletypewriter adapter in a diagnostic wrap state.

In the diagnostic wrap state, commands can be issued to the teletypewriter adapter for testing purposes. If a Write command is issued, data is sent to the teletypewriter adapter transmit data register and to the attached device, if present.



Using both diagnostic and input/output commands permits both online and offline checking of some devices. The system supplies diagnostic software for standard devices and interfaces, but user-written software for OEM devices can be similarly prepared and used.

Figure 138. Teletypewriter interface design

At the completion of the transmit operation, a device end interrupt is reported. The data is also sent to the teletypewriter adapter received data register; at the completion of the receive operation, an attention interrupt is reported. For checking purposes, the system can force the teletypewriter adapter into an overrun condition by: 1) not reading the received data register after the attention interrupt is accepted, and 2) then issuing another Write command. The teletypewriter adapter does not report condition code 1 (busy) or condition code 5 (interface data check) to this command.

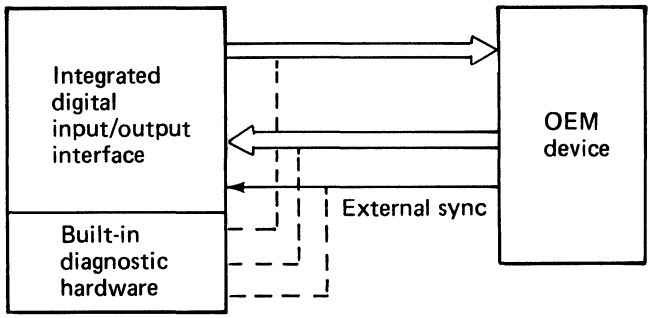
Exit from the diagnostic wrap state is by any of the following commands: a Device Reset, Halt I/O, System Reset, or Power-On Reset.

Diagnostic Commands

The integrated digital input/output interface provides similar capabilities (Figure 139). The system offers two commands to thoroughly test the correct operation of this interface.

The Set Test 0 command sets a diagnostic mode that disables the user inputs, including external sync. The ready line is disabled. The command places zero bits into the digital input receivers and activates the external sync receiver with a pulse. If external sync is armed, an interrupt is posted. The digital input data register contains all zeros. The user's previous data inputs and intervening commands govern the data in the process interrupt data register. If an interrupt is pending, condition code 1 (busy) is reported and the command is not executed. Also, when condition code 5 (interface data check) is reported, the command is not executed.

The Set Test 1 command sets a diagnostic mode that disables the user inputs, including external sync. The ready line is disabled. The command places one bits into the digital input receivers and activates the external sync receiver with a pulse. If external sync is armed, an interrupt is posted. The DI data register contains all one bits. The data in the PI data register is initially all one bits and, thereafter, is governed by intervening commands. If an interrupt is pending,



Diagnostic test commands permit:

- The setting of inputs at either zero or one levels
- Testing the external synchronization lines

Using these instructions, diagnostic software can, in effect, simulate the operation of an external OEM device; this insures that data is transmitted and received correctly, and that timing signals are properly armed and recognized.

The integrated digital input/output interface is used to interface OEM and special devices to the Series/1, and is fully supported from a diagnostic point of view. Self-diagnosis of the OEM device itself is the responsibility of the device designer or the system integrator.

Figure 139. The integrated digital input/output interface

condition code 1 (busy) is reported and the command is not executed. Also, when condition code 5 (interface data check) is reported, the command is not executed.

Other features have similar instructions and utilize the external wrap connections to improve problem analysis and failure diagnosis.

The reader should note that the diagnostic instructions are part of the overall system design. Any OEM device that interfaces to the system can have the same self-diagnostic capability as the system itself, but the OEM manufacturer must design this capability into the device, its interface, and its supporting software. This fact is important in those applications

where the user must add special devices because the system can support the added device in both its normal and abnormal operations—a major consideration in critical applications.

Error Logging

During a normal operation, Series/1 system software creates error logs to help in diagnosing problems. Utility software furnishes a dump of these logs which alerts maintenance and customer personnel to marginally-operating equipment. IBM has built other diagnostic aids, like communications' interfaces, into difficult-to-diagnose equipment. For example, the input/output and communications' facility trace functions are designed to continuously record current activity in main storage during normal operation of the system. The system uses this facility to reconstruct the sequence of events leading up to a system failure; the user can then more readily diagnose intermittent failures and other difficult system problems.

The communications' online test capability can test attached asynchronous terminals concurrent with user operation, and determine proper operation of the communications' link (lines and modems) as well as the terminal and system programming support.

These Series/1 diagnostic features are far more sophisticated than any previously available on small computers; today's demands for system availability require that these diagnostics be an integrated part of the system design. The extensive service aids provided with the Series/1 enable users to determine, by themselves, the source of many system problems.

Support for Maintenance

Most users are concerned about the support available for service and maintenance. The OEM users who provide their own processor and devices' support have access to all of the:

- Training courses
- Diagnostic software

- Maintenance consoles
- Signal tracing devices
- Other aids which IBM makes available to its own systems and customer engineers

These aids include full documentation and training in both IBM-supplied hardware and software. As discussed in Chapter 1, if users are to take responsibility for the system, they must have all the available information about the system as well as full training on the system.

For users who do not wish to undertake this responsibility themselves, IBM offers various contractual arrangements so that trained and available customer engineers can provide the proper level of service. This availability of highly trained and knowledgeable—hardware and software—service people is a very valuable backup to the OEM or third-party system integrators who occasionally need in-depth backup to solve a particularly critical problem with one of their systems. The importance, to users and suppliers, of available, high-quality, trained personnel cannot be overestimated.

In summary: the IBM Series/1 is an integrated design of hardware, software, and maintenance designed to provide a set of modules or tools which can be combined with user- or other vendor-supplied modules to build an economical, small computer application. The integrated design of the system insures that:

- Applications can be performed
- Implementation can be controlled
- Errors can be diagnosed
- Overall system availability can be assured

These capabilities are the primary prerequisites for a satisfactory application.

Index

International Business Machines Corporation

General Systems Division
4111 Northside Parkway N.W.
P.O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
(International)

SH30-0237-0
Printed in U.S.A.