NonStop™ Systems

System Description Manual

Operating System Library

82507

NonStop™ Systems

System Description Manual

Abstract
This manual provides architectural descriptions of the Tandem NonStop II™ and NonStop TXP™ processor hardware and the GUARDIAN™ operating system.

Product Version
NonStop II and NonStop TXP Processors
GUARDIAN B00 operating system

Operating System Version
GUARDIAN B00 (NonStop Systems)

Part No. 82507 A00

March 1985

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599

NOTICE

Effective with the B00/E08 software release, Tandem introduced a more formal nomenclature for its software and systems.

The term "NonStop 1+™ system" refers to the combination of NonStop 1+ processors with all software that runs on them.

The term "NonStop™ systems" refers to the combination of NonStop II™ processors, NonStop TXP™ processors, or a mixture of the two, with all software that runs on them.

Some software manuals pertain to the NonStop 1+ system only, others pertain to the NonStop systems only, and still others pertain both to the NonStop 1+ system and to the NonStop systems.

The cover and title page of each manual clearly indicate the system (or systems) to which the contents of the manual pertain.

DOCUMENT HISTORY

Edition	Part Number	Operating System Version	Date
1st Edition	82077 A00	GUARDIAN A00	April 1981
2nd Edition	82077 B00	GUARDIAN A03	April 1982
3rd Edition	82077 C00	GUARDIAN A04	October 1982
4th Edition	82077 D00	GUARDIAN A06	December 1983
5th Edition	82507 A00	GUARDIAN B00	March 1985

New editions incorporate all updates issued since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages that you should merge into the most recent edition of the manual.

The part number of this manual changes with this fifth edition. This change was made to accommodate two current versions of the manual while the B00 software is in limited release.

The following are trademarks or service marks of Tandem Computers Incorporated:

AXCESS	BINDER	CROSSREF	DDL	DYNABUS
DYNAMITE	EDIT	ENABLE	ENCOMPASS	ENCORE
ENFORM	ENSCRIBE	ENTRY	ENTRY520	ENVOY
EXCHANGE	EXPAND	FOX	GUARDIAN	INSPECT
NonStop	NonStop 1+	NonStop II	NonStop TXP	PATHWAY
PCFORMAT	PERUSE	SNAX	Tandem	TAL
TGAL	THL	TIL	TMF	TRANSFER
T-TEXT	XRAY	XREF		

INFOSAT is a trademark in which both Tandem and American Satellite have rights.

HYPERchannel is a trademark of Network Systems Corporation.

IBM is a registered trademark of International Business Machines Corporation.

NEW AND CHANGED INFORMATION

This manual is the fifth edition of the System Description Manual for NonStop systems. It includes the following changes to the fourth edition:

- Sections 1 and 2 have been revised and reorganized to improve the system introduction.
- Sections 4 and 5 have been revised to reflect the multisegment address space capability provided in the B00 version of the operating system.
- Instruction definitions in Section 9 have been expanded to document the microcode support for the B00 operating system.
- Two new sections, 10 and 11, have been added. These provide an introduction to the internals of the GUARDIAN operating system.
- Minor technical and typographical errors have been corrected.

CONTENTS

FIGURES

TABLES

PREFACE

This manual provides a conceptual and functional description of Tandem NonStop systems, which can be composed of NonStop II and/or NonStop TXP processors, and the GUARDIAN operating system. The manual content is presented as follows:

- Section 1 provides an overview of the Tandem NonStop system, introducing both the hardware architecture and the GUARDIAN operating system.
- Section 2 describes the principles on which the hardware and firmware operate and shows how these principles support the NonStop system architecture.
- Section 3 describes the data formats and the number representation used for the NonStop II and NonStop TXP processors.
- Section 4 describes program execution from the hardware standpoint.
- Section 5 describes addressing and memory access from a hardware viewpoint for the NonStop II and NonStop TXP processors.
- Section 6 describes the hardware aspect of the Interrupt System.
- Section 7 describes the interprocessor buses and the input-output channel.
- Section 8 describes cold load (I/O cold load and bus cold load) for the NonStop II and NonStop TXP processors.
- Section 9 defines the instruction set for NonStop II and NonStop TXP processors in text form with illustrations.

- Section 10 describes the primary components and structures of the GUARDIAN operating system.
- Section 11 describes the environment and fundamental attributes of processes.
- Appendixes A and B consist of reference tables pertaining to the instruction set.
- Appendix C provides a high-level comparison of three different processors manufactured by Tandem Computers: the NonStop 1+ processor, the NonStop II processor, and the NonStop TXP processor.
- An index is provided to assist the reader in locating specific topics in this manual.

This manual was written for potential and present Tandem customers seeking a functional description of the system hardware and the operating system, for Tandem field analysts and service engineers, and for students in various courses provided by Tandem.

Before using this manual, you should read Introduction to Tandem Computer Systems for a more general overview. The introductory manual explains the basic concepts and purposes behind the system architecture described in this manual. Ideally, you should also have some working experience with Tandem systems.

SECTION 1

INTRODUCTION

HARDWARE SYSTEM STRUCTURE

The Tandem NonStop computer system is designed to provide continuous operation, incorporating fault-tolerant features in all levels of the system structure. Significantly, the hardware and firmware components are designed to allow both continued execution of processes and continued access to data bases when a system component fails. These design goals are illustrated in diagram 1 of Figure 1-1.

Fault tolerance for system and user processes is accomplished by executing a secondary (or "backup") process in another processor, programmed to require only periodic "checkpoint messages" to keep up to date on the current state of the primary process. Upon any failure of the processor that is executing the primary process, the backup process can resume execution of the work from the point of the last valid checkpoint. The backup process, instead of the primary process, will then be accessing the data base on disc. As indicated in the diagram (1), dual data paths between processors assure communication of the checkpoint messages.

Fault tolerance for the user's data base is accomplished primarily by the use of dual-ported controllers and, optionally, by maintaining duplicate data on two separate disc volumes ("mirrored" volumes). For mirrored volumes, all data written out to the user's files is automatically written into both disc volumes. Thus, whenever data is read from the files, either volume can be accessed, since they contain identical information. As in the case of interprocessor communication, two data paths to the disc volumes are provided.

The various hardware features that accomplish these design goals are considered under the following subheadings--illustrated by the remaining diagrams in Figures 1-1, 1-2, and 1-3.

It should be noted in considering the following information that, although the mechanics for switching between multiple modules and data paths reside in the hardware, the control of such actions is a function of the GUARDIAN operating system.

Independent Multiple Processors

The Tandem NonStop system consists of 2 to 16 processor modules. A processor module is sometimes referred to as a central processing unit, or CPU for convenience, although in a Tandem system, no one processor is more "central" than any other. Each processor (CPU) contains the functions that normally comprise a complete computer system: instruction processing unit (IPU), memory, and input-output channel. In addition, each module contains logic for a fourth main function: the interprocessor bus interface through which the processors communicate with each other. Furthermore, each module is associated with its own separate power supply. (See diagram 2 in Figure 1-1.) Therefore, each processor module is capable of operating independently of, and simultaneously with, all other processor modules in the system.

This fundamental design feature means that each processor is totally self-sufficient. An IPU failure, for example, cannot prevent another processor from functioning, since there are no shared elements such as memory. A failing IPU cannot contaminate any memory data outside of its own module.

Dual-Bus Data Paths

Each processor module is connected to all other processor modules through redundant high-speed interprocessor buses, each controlled by its own separate bus controller. See diagram 3 in Figure 1-1. Programs running in one processor module communicate with programs running in other processor modules by means of these buses. Each interprocessor bus is fully autonomous, operating independently of (but simultaneously with) the other bus.

The use of two buses assures that two paths exist between all processor modules in the system. If one bus fails, all interprocessor communication is automatically routed over the remaining bus. The use of bus controllers that are separate and independent of the logic circuits within the processors assures that no failure of a processor module will cut off bus transmission.

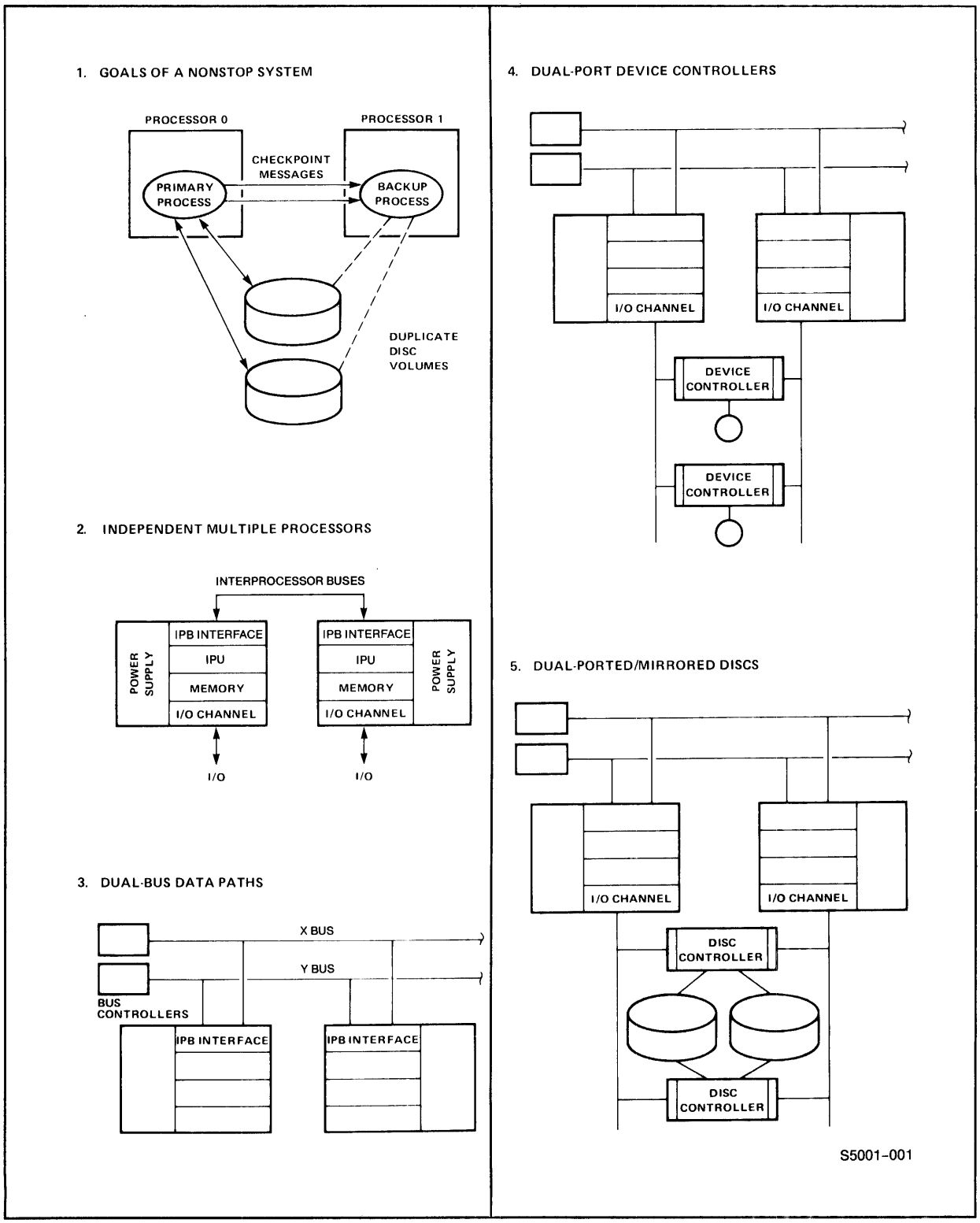


Figure 1-1. Elements of Hardware System Structure

The interprocessor bus interface in each module is capable of accepting transmissions from either bus, under control of the operating system.

Dual-Port Device Controllers

Data is transferred between an input-output device (such as a disc, terminal, or line printer) and a processor module by means of an input-output channel. Each processor module has one I/O channel that is capable of communicating with up to 256 I/O devices. See diagram 4 in Figure 1-1.

I/O devices are interfaced to the I/O channels by dual-port controllers. Each dual-port controller is connected to the I/O channels of two processor modules. Therefore, each I/O device can be controlled by either of two processor modules. However, in operation, an I/O device is controlled exclusively by one processor module until a failure occurs such that the processor module can no longer communicate with the I/O device. If such a failure occurs, the other processor module takes control of the I/O device.

Dual-Ported and Mirrored Discs

Because discs represent the most critical class of I/O devices, disc drives can also have dual ports. In combination with the dual ports on the disc controller, various configurations are possible to meet any desired degree of fault tolerance. For example, connecting the dual ports of the controller to separate I/O channels provides for fault tolerance of the I/O channels. Connecting dual ports of a disc drive to separate controllers provides for fault tolerance of the disc controllers. Diagram 5 of Figure 1-1 shows an example of a fully mirrored, fully dual-ported configuration.

Multiple Power Sources

Power is distributed in the system in such a manner that each dual-port controller receives power from two sources. If a supply fails, causing a processor module to become inoperative, the alternate power supply can assume the full load.

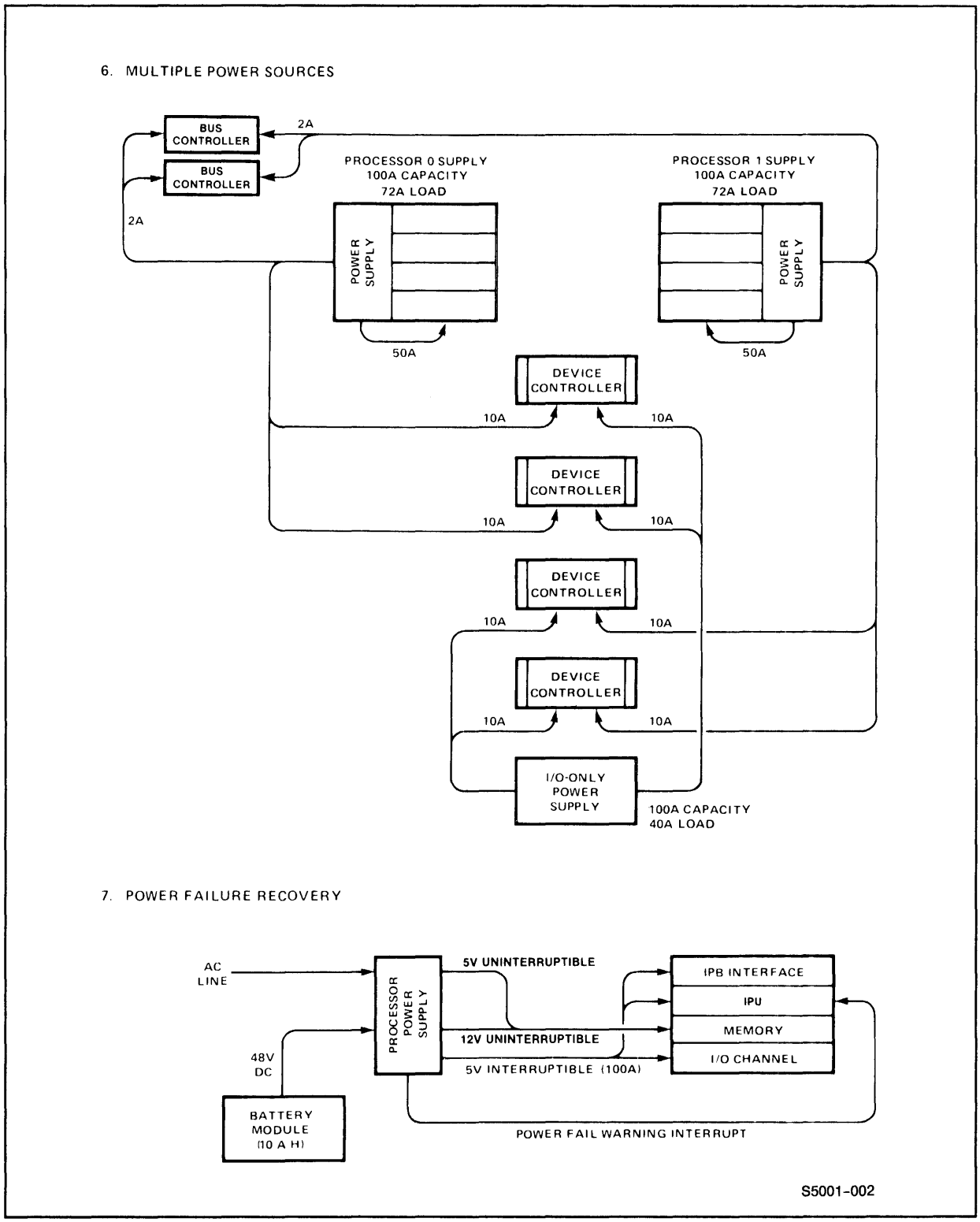


Figure 1-2. Power Distribution for NonStop II Processors

In a NonStop II processor, the processor consumes approximately half the power available from its supply; the remainder is available to help power the device controllers. In some cases, the power available from these supplies is sufficient to power all the device controllers; in other cases, a supplementary power supply for I/O only is necessary.

In a NonStop TXP processor, the processor consumes most of the power available from its supply--the CPU power supply is not available to help power the device controllers. The device controllers must receive their power from an I/O-only power supply.

Diagram 6 in Figure 1-2 shows, in simplified form, the way in which power is distributed in a system using NonStop II processors in order to achieve reliable power backup. The current values shown are mostly illustrative only; device controllers, for example, generally take much less than the 20 amperes assumed in this figure. Exact values and the adjustments required to achieve good power distribution are evaluated by Tandem for each particular system when the system is configured.

As shown, the two bus controllers require a total of about 4 amperes, 2 amperes each from the supplies associated with processor 0 and processor 1. (Bus controller power is always taken from the supplies for these particular CPUs.) The processor modules are assumed to require 50 amperes each; this depends on memory size and configuration. The output current capacity of the supplies is 100 amperes each (for the 5-volt interruptible supply, discussed later). Note that each device controller nominally receives one-half of its requirements (10 amperes) from each of two different power supplies. (In actuality, adjustments are made so that the CPU supply provides somewhat less than half the needed power, and the I/O supply provides slightly more than half.) Under the assumed conditions, then, each processor's power supply is loaded to 72 amperes, and the I/O-only supply is loaded to 40 amperes.

Now assume a failure in the processor 0 power supply. The processor 0 module goes down, but none of the device controllers or bus controllers is affected. The processor 1 power supply now delivers the full 4 amperes needed by the bus controllers (increasing its load to 74 amperes), and the I/O-only power supply delivers the full 20 amperes to each of the uppermost two device controllers (increasing its load to 60 amperes).

Likewise, if the I/O-only power supply should fail, the load on each processor's power supply increases by 20 amperes (to 92), still within the 100-ampere capacity. Thus, any single power supply failure can be compensated by increased loading on the remaining supplies. However, the failure of any two supplies cannot always be accommodated by the remaining ones.

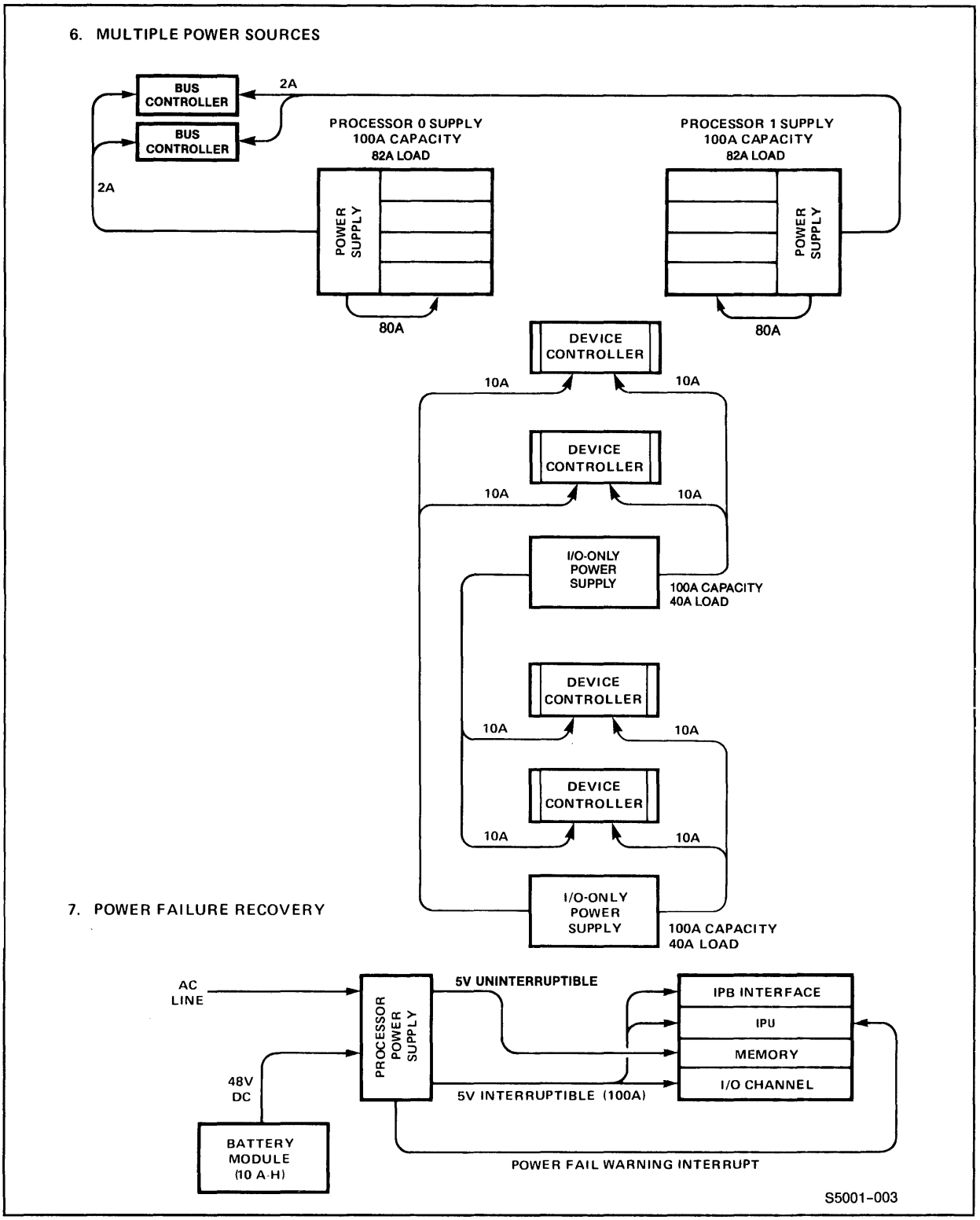


Figure 1-3. Power Distribution for NonStop TXP Processors

Diagram 6 in Figure 1-3 shows, in simplified form, the way in which power is distributed in a system using NonStop TXP processors. The current values shown are illustrative only.

Again, the two bus controllers require a total of about 4 amperes, 2 amperes each from the supplies associated with processor 0 and processor 1. The processor modules are assumed to require 80 amperes each; this depends on memory size and configuration. The output current capacity of the supplies is 100 amperes each (for the 5-volt interruptible supply, discussed later). In this example, each device controller now receives one-half of its requirements (10 amperes) from each of two different I/O-only power supplies.

Now assume a failure in the processor 0 power supply. The processor 0 module goes down, but neither the device controllers nor the bus controllers are affected. The processor 1 power supply now delivers the full 4 amperes needed by the bus controllers (increasing its load to 84 amperes). Power distribution to the device controllers remains unaffected.

Likewise, if an I/O-only power supply should fail, the load on the remaining power supply increases by 40 amperes (to 80), still within the 100-ampere capacity. Thus, any single power supply failure can be compensated by increased loading on the remaining supplies. However, the failure of any two supplies cannot always be accommodated by the remaining ones.

Power-Failure Recovery

Diagram 7 in Figures 1-2 and 1-3 illustrates the power-failure recovery features that are incorporated into the internal circuits of each processor module. Note that memory is powered separately from the rest of the module, with its own 5-volt and 12-volt supplies (memory for a NonStop TXP processor does not require 12 volts); these are termed uninterruptible supplies, since they are maintained by battery power if an AC line failure occurs. Battery power then allows memory to retain its contents for 1.5 hours or more, depending on memory size and the charge state of the battery.

The interruptible 5-volt supply powers the remainder of the module. In order to allow the operating system to bring the CPU to an orderly halt, the power supply issues a special signal (power fail warning interrupt) when AC power is lost for more than 24 milliseconds. This signal gives a minimum of 5 milliseconds warning (depending on loading of the supply) that the 5-volt supply is going down.

The system automatically restarts upon restoration of power, resuming execution of the processes that were in progress at the time of the power failure.

Other Reliability Features

The ability of the Tandem NonStop computer system to provide an environment where applications can continue to run regardless of a module failure is due primarily to its unique fault-tolerant features, described above. In addition to those unique features, the NonStop system also incorporates various other reliability features, which include the following:

- In the event of a power failure, each processor module (under control of its operating system) saves its current operating state in memory. When power is restored, the hardware automatically invokes the appropriate operating system procedures to resume all operations.
- If an uncorrectable error occurs in memory, the operating system determines if the associated area is critical to system operation. If it is not, the area is flagged as bad and not used again until the memory is repaired. If the area is critical, the operating system halts execution in its processor.
- Critical portions of the operating system are main-memory resident; this assures their availability in the event a virtual memory (disc) failure occurs.
- The cooling system for the computer is designed so that if a single failure occurs, ample cooling is still available.
- Any module in the system (i.e., processor, I/O controller, power supply, fan, etc.) can be removed from the system and replaced online without stopping operation of other system modules.
- Routing, sequence, and checksum words are generated by the transmitting processor module and checked by the receiving processor for every packet of 13 data words transferred over the interprocessor buses.
- A parity bit is associated with each 16-bit word transmitted over the I/O channels.
- An interval timer is provided; the operating system uses the timer to notify the application program in the event a data transfer does not complete.

- Six error correction bits are generated and stored with each 16-bit word in the semiconductor memory; circuitry is provided to correct all single-bit errors and detect all double-bit errors.

- The addressing and count information associated with I/O transfers is kept in the controlling processor module. This prevents a controller from contaminating more than one processor module because of a failure of an address or word count register.

- The memory mapping scheme provides separate system and user address spaces. Operating system data areas can be accessed only by operating system programs; application programs cannot inadvertently destroy the operating system.

- Parity checking is provided for the NonStop II processor's memory map registers.

- Parity checking is provided in the NonStop TXP processor's caches.

- Two hardware modes of processor operation are provided: privileged and nonprivileged. Certain critical operations (such as accessing system tables from application programs or initiating input-output transfers) can be performed only while in privileged mode. Typically, only the GUARDIAN operating system runs in privileged mode; privileged operations are performed on behalf of application programs through calls to operating system procedures. Application programs running in nonprivileged mode are prevented from becoming privileged.

OPERATING SYSTEM OVERVIEW

In the Tandem NonStop computer system, master copies of the GUARDIAN operating system software are maintained in the system subvolume of a mirrored disc volume (Figure 1-4); each CPU uses or executes appropriate portions of this master copy, depending on its unique configuration. The mirrored system volume is a pair of physically independent disc devices, usually attached to separate controllers but accessed as a single volume and managed by the same executing input-output program.

Critical and frequently used parts of the operating system reside permanently in each CPU's memory. Noncritical or less frequently used portions reside in virtual memory; they are brought into CPU memory from disc only when needed, by way of the CPU (one of two) that is currently controlling the system volume. The duplication of GUARDIAN software, plus the fact that there is a dual path to

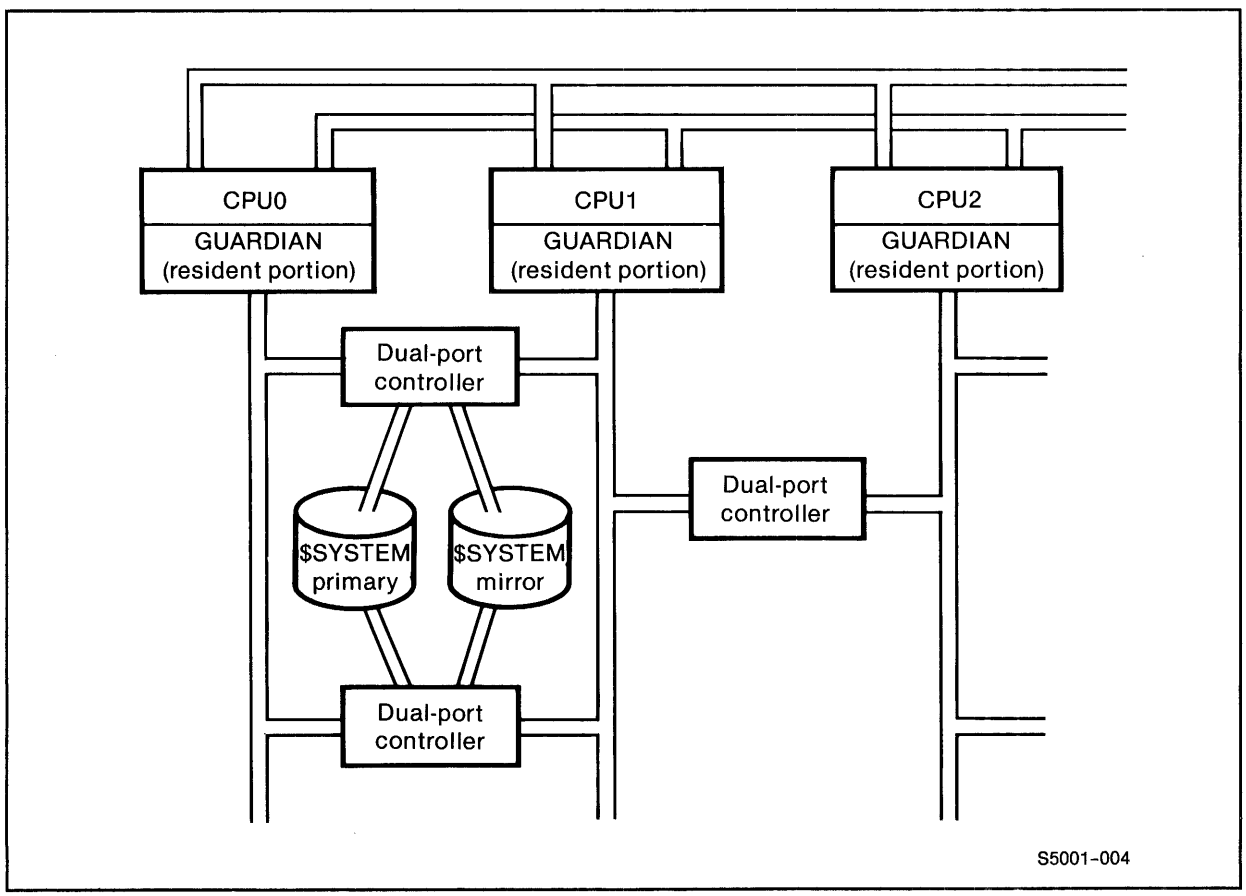


Figure 1-4. Tandem Computer System Failure-Tolerant Environment

the system volume, guarantees coninued system operation even if a CPU, input-output channel, or disc drive fails.

Normal GUARDIAN software operation frequently requires that CPUs in the system depend on one another (Figure 1-5). For example, the virtual memory disc input-output done for a process in CPU 0 may actually be performed by a disc process in CPU 2.

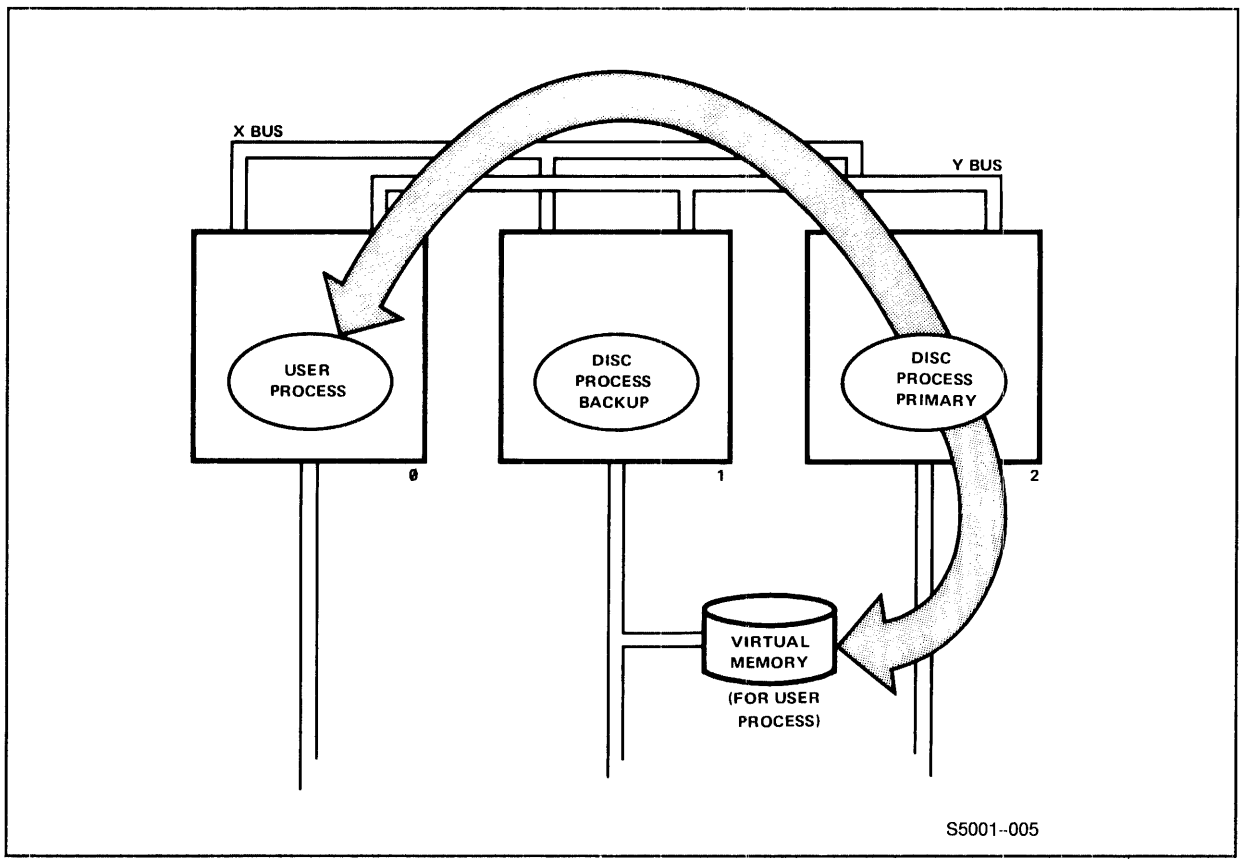


Figure 1-5. Interprocessor Dependency

Thus, although each CPU essentially operates independently under control of its own operating system, all CPUs need to be able to communicate with each other. To provide a reliable basis for this interprocessor communication, each CPU monitors the status of all other CPUs in the system. If a particular CPU ceases to operate as it should, early detection of the failure and prompt notification of any processes that back up those in the malfunctioning CPU allow the system to continue operating. To detect such problems, the GUARDIAN software uses messages

transferred over the interprocessor bus (Figure 1-6). In this message scheme, every CPU in the system must receive a message from all other CPUs (as well as itself) at least once during a predetermined polling period of approximately one second. For this reason, each CPU transmits messages to indicate that it is still operating. Such messages are called "I'm alive" messages.

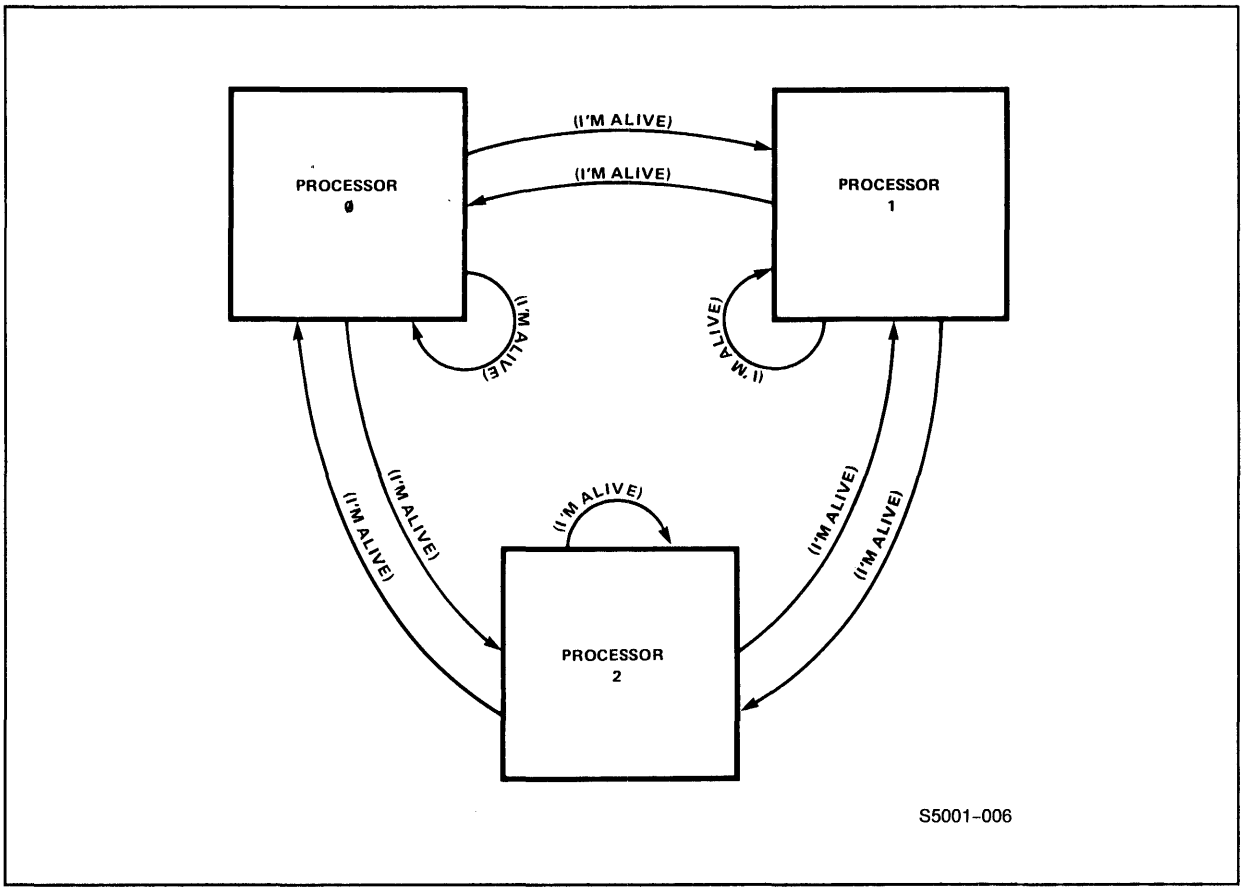


Figure 1-6. Failure-Detection Messages

If a CPU does not receive any "I'm alive" messages from a particular CPU during two consecutive polling periods, it declares that CPU down. If a CPU does not receive its own "I'm alive" message, it continues to operate but does not make any attempt to take ownership of any terminals, disc drives, or other devices. Usually, a CPU that does not respond has failed one of the many internal consistency checks that the operating system regularly performs. (Less likely, though also possible, is a failure of one of the interprocessor buses.) A serious failure

causes the CPU to halt, with the halt reason indicated to the system operator. This prevents the CPU from sending "I'm alive" messages; as a result, this CPU is soon declared down by all other CPUs in the system.

MAIN OPERATING SYSTEM COMPONENTS

The GUARDIAN operating system contains four logically distinct areas:

- User-callable library procedures (and associated routines)
- System processes (including associated data segments)
- Kernel
- System data structures

A conceptual view of these areas and their interrelationships appears in Figure 1-7.

In this figure, oval symbols depict both system and user processes. The two octagonal shapes represent portions of the system library, which consists of user-callable library procedures and kernel procedures. Pairs of dotted lines show paths between the process and library elements of the diagram; these paths illustrate the information flow within the system. In some cases, the paths are traversed by procedure calls and returns. In other cases, they actually carry messages to and from processes. But in either event, they illustrate an ability to make and satisfy requests--to pass and return information.

The natural focal points of the system, as illustrated in Figure 1-7, are the user-callable library and the kernel. Requests generated by application processes focus on user-callable library procedures--they form the application process's window to the system. Communication paths between system processes, however, focus naturally on the kernel portion of the system library.

A short discussion of these and other components appears under the following subheadings.

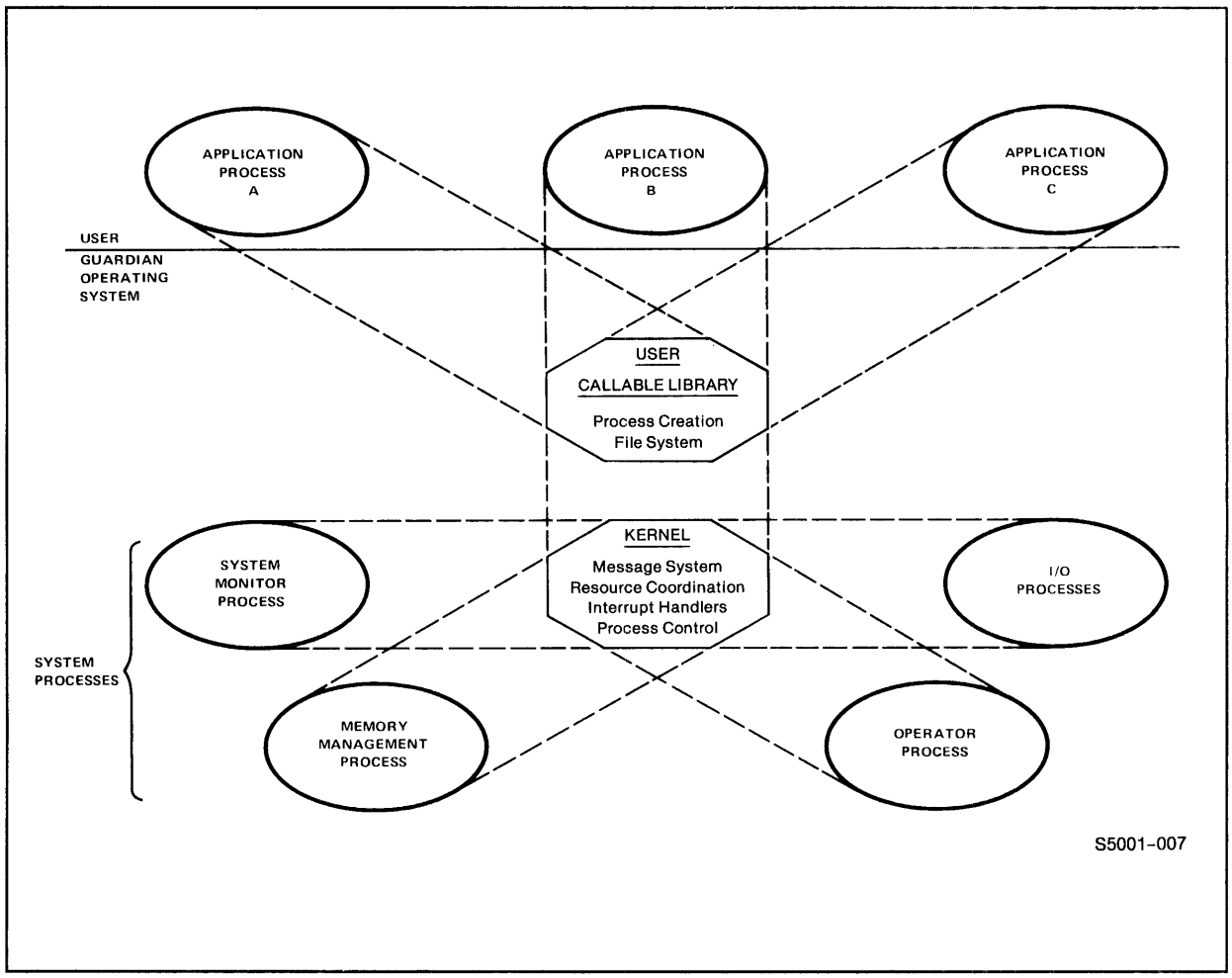


Figure 1-7. Logical Operating System Components

User-Callable Library Procedures

Executing application processes request operating system services by issuing calls to the user-callable library procedures (or simply "callable" procedures). These procedures operate in the data environment of the calling process. They use the process's data area as their temporary storage space, but their privileged-mode execution also gives them access to the system table structures, such as those stored in the system data segment (Figure 1-8).

User-callable library procedures, such as OPEN, READ, WRITE, and CLOSE, are located in the system code area and so can be shared by any and all processes that need the services they provide.

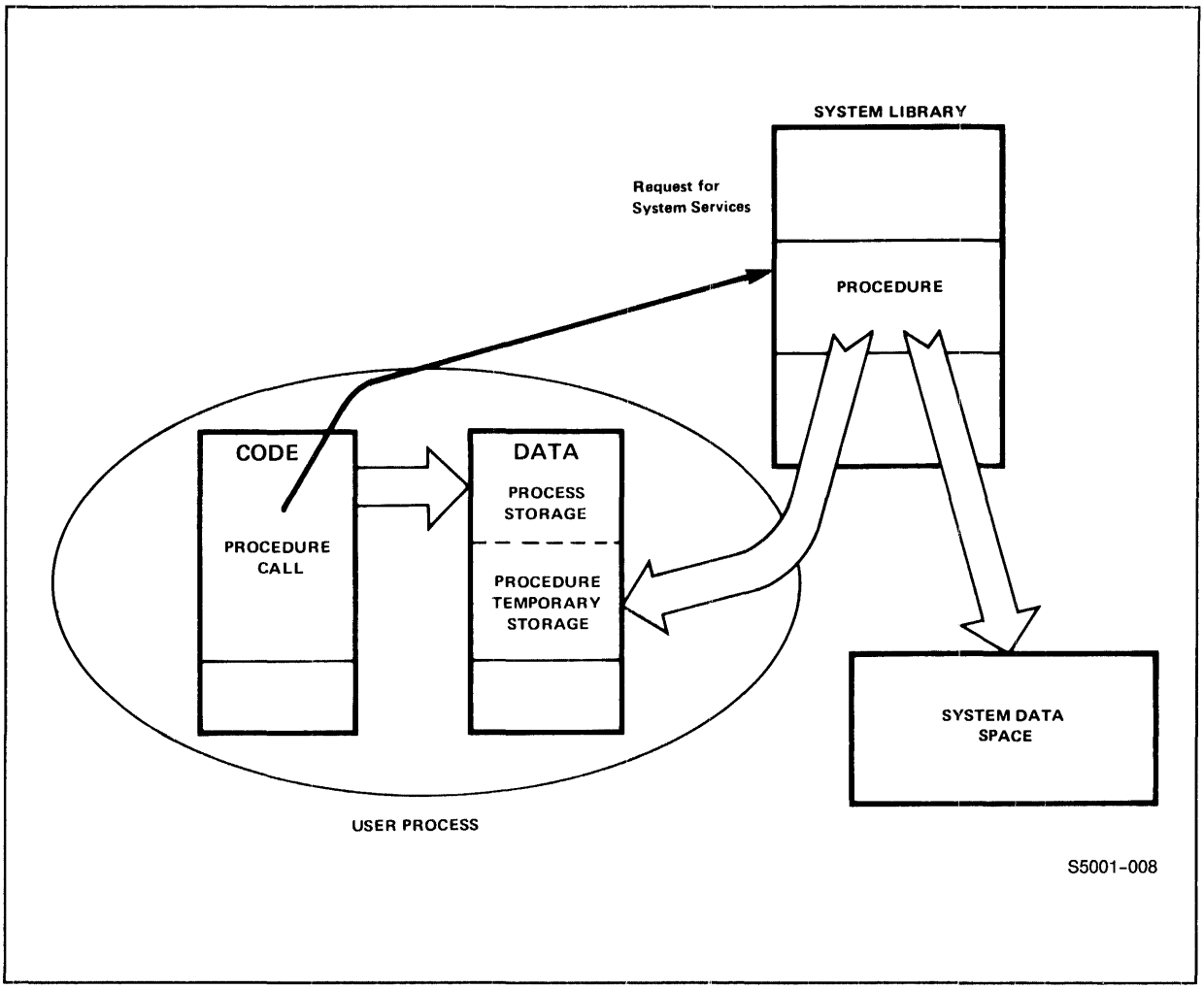


Figure 1-8. Application Process Access to System Services

System Processes

System processes constitute a limited set of privileged processes that come into existence at cold-load time and exist continuously for a given configuration as long as the host CPU remains operable. The system processes primarily consist of a memory manager and a monitor in each CPU, and operator and I/O processes distributed in various CPUs of the system. Each I/O process pair logically "owns" one or more I/O devices, and in order to access these devices, other processes must sent a request to the "owning" process. If the owning process decides to honor the request, it will provide the necessary service and return a "reply" to the requesting process.

The location of the various system processes in a three-CPU system is shown in Figure 1-9. Notice that some of the processes are present in every CPU, but others (mainly those related to input-output) are found only in the CPUs connected to their

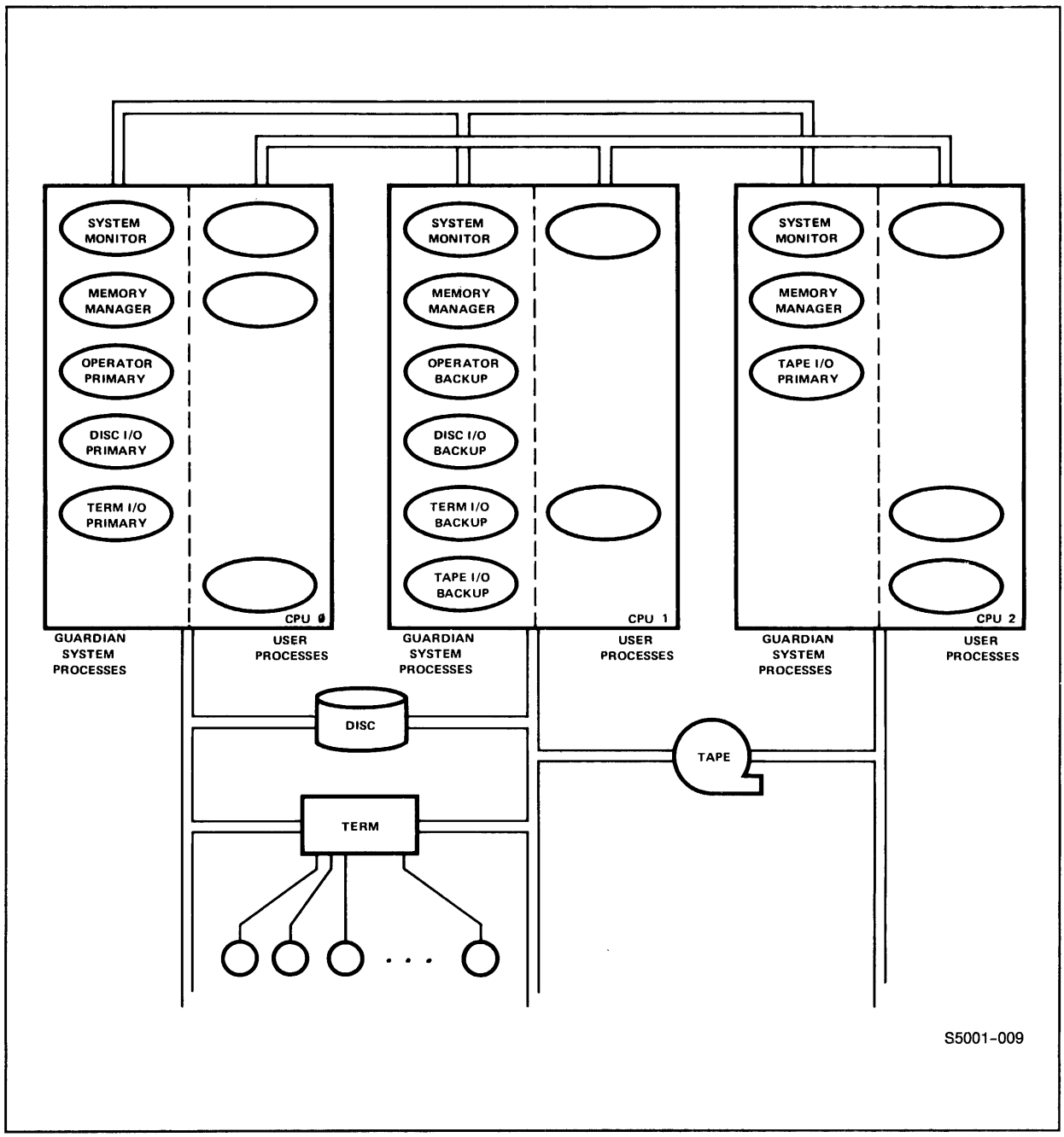


Figure 1-9. Distribution of System Processes

associated peripheral devices. Actual determination of these locations is dependent on system configuration, and is specified at system generation time. The following paragraphs describe the main functions of these system processes.

Memory Manager. The memory manager (Figure 1-10) services requests generated by interrupt handlers as well as by other system processes. Primarily, this process implements the paging scheme for virtual memory.

The memory manager receives special requests from the Page-Fault interrupt handler to bring needed pages into CPU memory from disc. It is also used by the monitor process to help set up the memory environment of a new process that is being created in the CPU. Because the memory manager deals only with memory resources in the CPU where it is running, a separate memory manager process must reside in every CPU in the system.

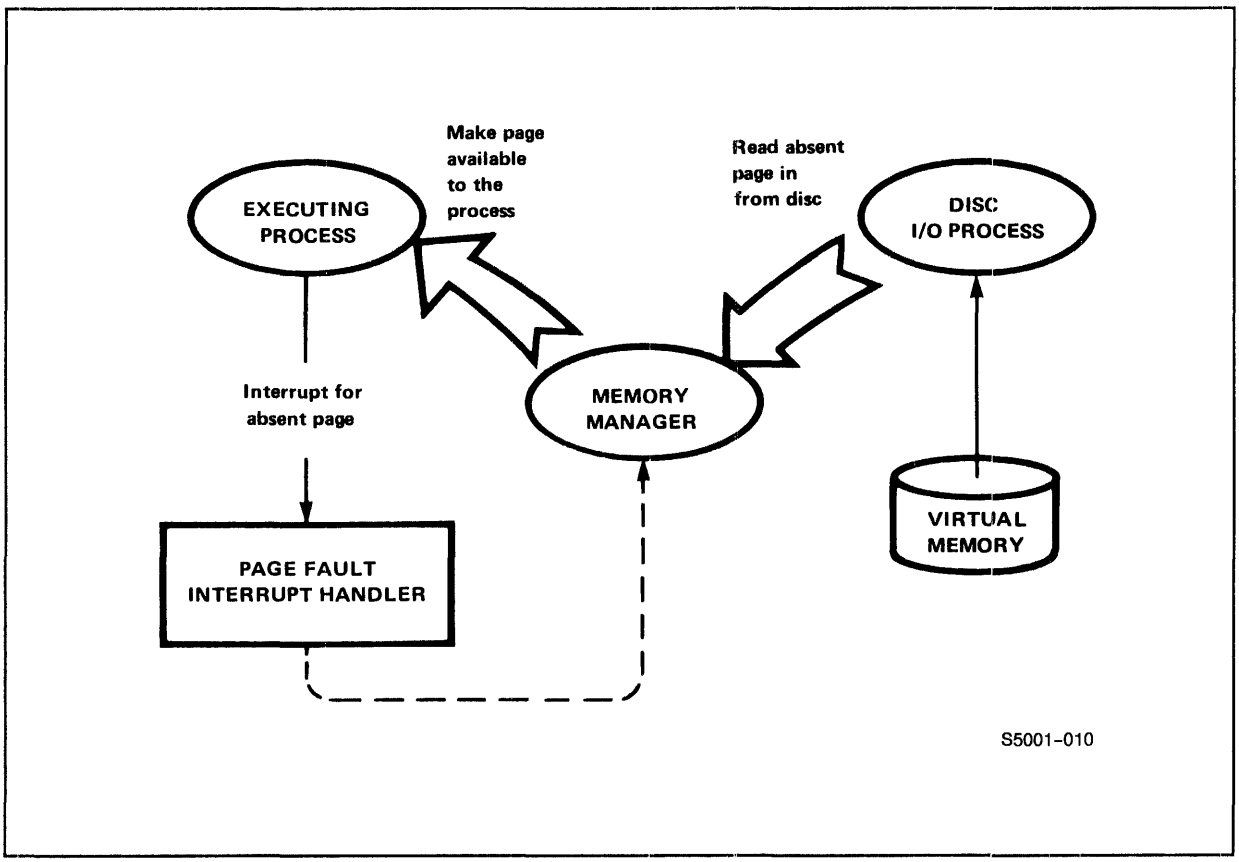


Figure 1-10. Memory Manager Process

Monitor. Like the memory manager, a monitor process runs in each CPU on the system (Figure 1-11). This process handles many housekeeping functions and initiates process creation and deletion done within its particular CPU. It also serves as an information source for processes running in all CPUs in the system. For example, if a process running in CPU 2 needs status information about a process running in CPU 0, the monitor in CPU 0 is contacted with a request to locate and return the necessary information.

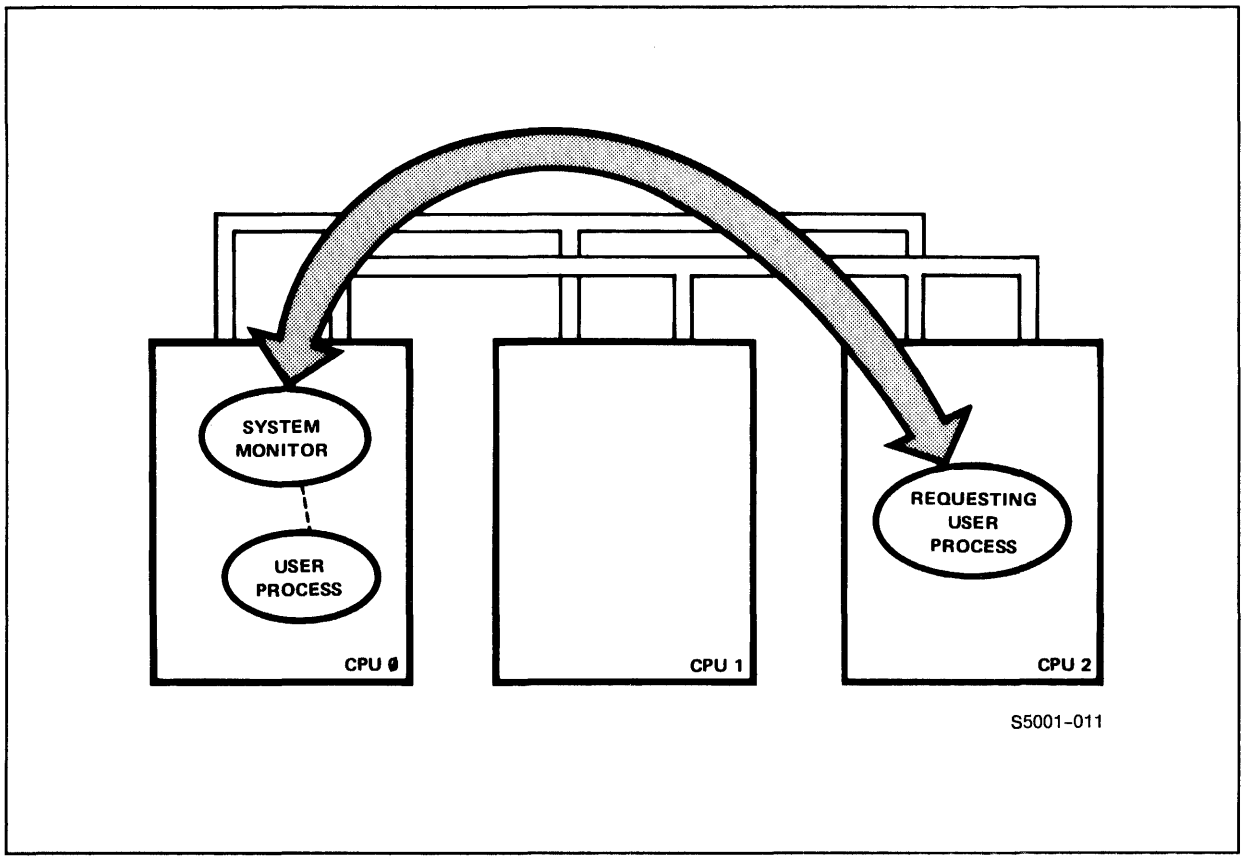


Figure 1-11. Monitor Process

Operator Process. Another system process, the operator process, runs as a process pair. Unlike the memory manager and the monitor, there are only two copies of this process (a primary and a backup) in the entire system. As illustrated in Figure 1-12, the main responsibility of the primary process is to transmit Operator process messages to the system console and to disc.

The backup process receives messages from the primary to inform it of actions in progress. This enables the backup to assume the duties of the primary if the primary fails.

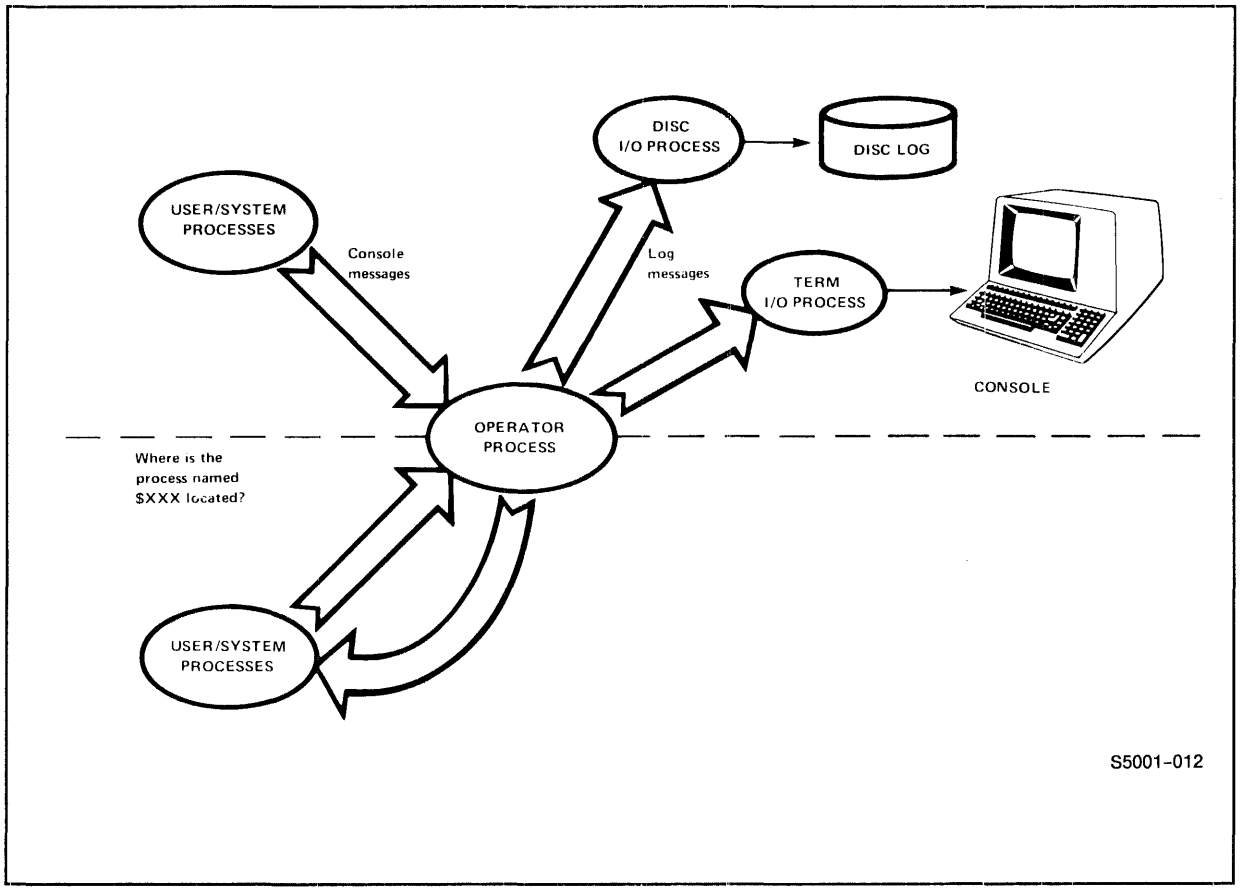


Figure 1-12. Operator Process

Input-Output Processes. These processes manage input-output hardware. Typically, an input-output process controls a single physical device. For instance, line printers, card readers, and unmirrored discs each have their own input-output processes. As exceptions to this rule, however, terminal processes, communications processes, and disc processes associated with mirrored disc volumes can each be called upon to control multiple physical devices. A copy of the input-output process for a particular device resides in the memory of each CPU connected to the device's controller. A process pair is actually involved in all input-output: the primary process is active in controlling the device while the backup process takes over if a CPU or input-output channel fails. These two processes run on separate CPUs, but use the same code. At critical points, the primary transmits its state and current data to the backup. This enables the backup process to continue the operations being done by the primary process if the primary is no longer able to function.

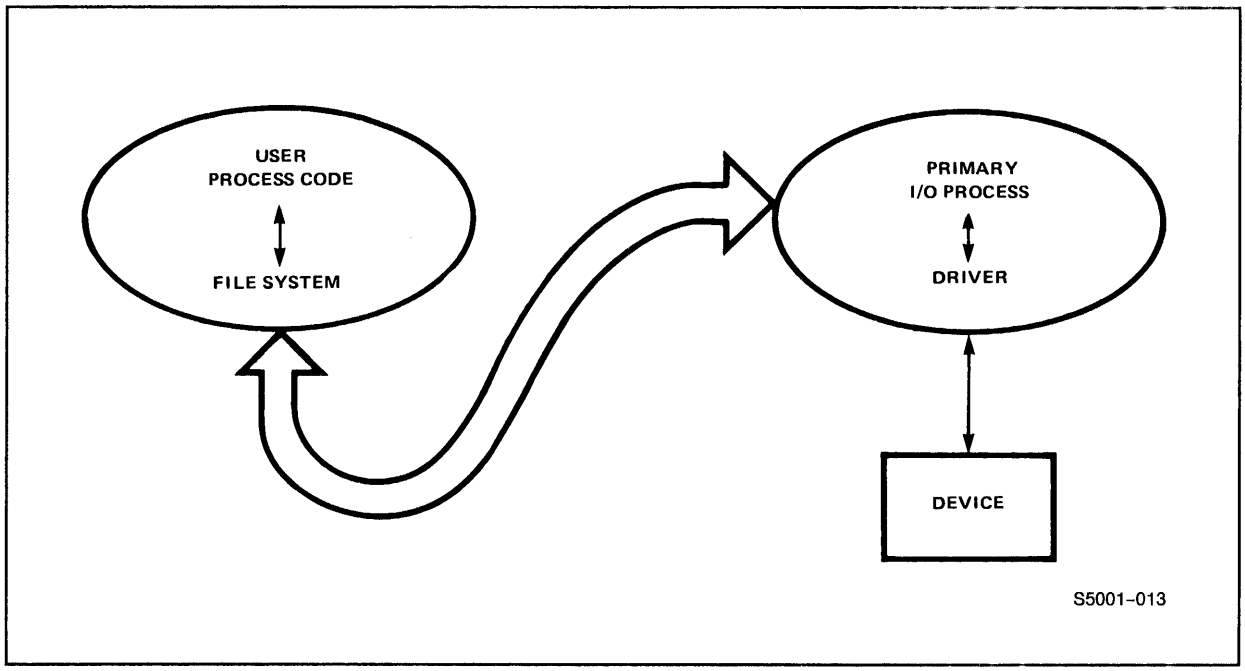


Figure 1-13. Input-Output Process

Requests for input-output operations usually come from the GUARDIAN file system. The file system is a set of system procedures, part of which are in the user callable group and part in the kernel. These procedures run as part of the user's process and send messages to the input-output process, which is responsible for controlling the device. The input-output process, in turn, calls its own procedures to deal with the device dependencies of the peripheral involved and to handle the physical transfer of data. Appropriate status and data values are returned to the user's process by reply messages (Figure 1-13).

Kernel

The kernel is a set of system library procedures that provide first-level software extensions to the basic hardware capability of the Tandem computer system. The kernel incorporates four types of low-level system operations:

- Interrupt handling
- Resource coordination (including counting semaphores and mutual exclusion)
- Interprocess message transfers
- Process management

Interrupt Handling. Some of the kernel procedures are invoked when an interrupt occurs. These interrupts can result from a number of causes, including: hardware errors, references to code or data pages absent from memory, completion of interprocessor bus messages, input-output transfer completions, timer list updates, and process execution requests.

In most instances, each action that can cause an interrupt corresponds to a particular bit in the Interrupt Request (INTA) register. For example, an input-output interrupt request generated by the hardware (standard I/O) always sets Bit 14 of this register. When the microcode that manages interrupts detects that one of these bits is ON, it checks to determine whether the corresponding bit in the Interrupt Mask register is also ON. If this matching bit is ON, the interrupt occurs; otherwise, the interrupt is postponed until the matching bit in the mask is set to allow it. For example, suppose that an input-output controller needs to request an interrupt. The following events take place (see Figure 1-14).

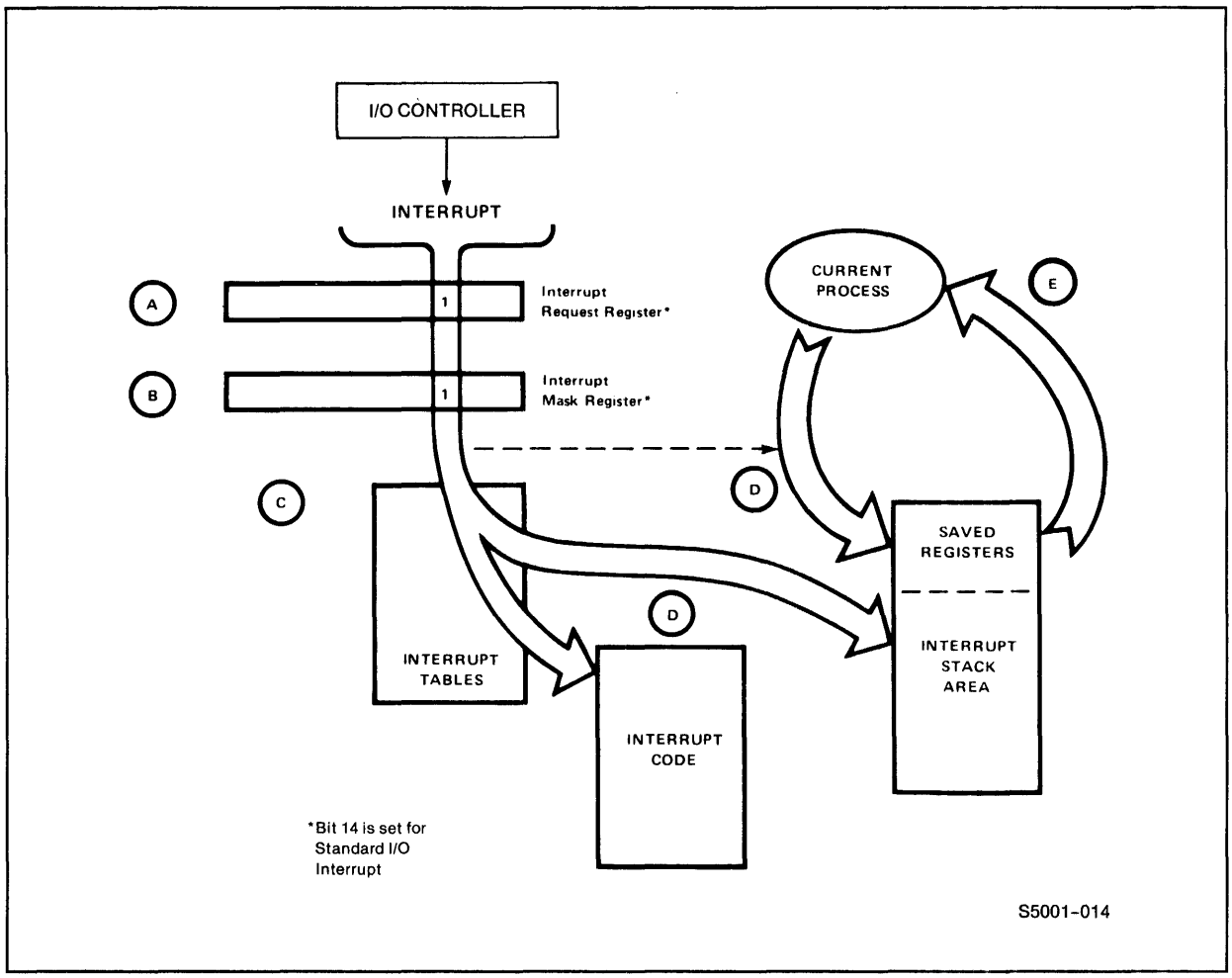


Figure 1-14. Interrupt Handling

1. The I/O controller sends an interrupt request to the IPU by way of the I/O channel. The IPU accordingly sets bit 14 in the Interrupt Request Register ON ("A" in Figure 1-14). When the interrupt-managing microcode next checks this register, it detects that Bit 14 is on.

2. Since Bit 14 of the Interrupt Mask register is also ON ("B" in Figure 1-14), the microcode begins to process the interrupt.

3. The microcode uses the number of the bit set in the Interrupt Request register (bit 14) as an index to an entry in a table in the system data segment ("C" in Figure 1-14). This entry supplies data on how the interrupt environment should be

established. It contains, among other elements, the addresses of the code and data to be used by the interrupt handler procedure.

4. The microcode saves the executing environment for the current process in the interrupt handler's data area and sets the code and data registers to define the interrupt handler's environment ("D" in Figure 1-14). Execution now continues in the interrupt handler's code.

NOTE

Although they use code and data, interrupt handlers are NOT processes. Unlike processes, they are invoked by hardware or microcode, and have no entries in the system tables associated with processes. The operating system maintains interrupt-handler code and data in main memory at all times because they must respond instantaneously to interrupts.

5. When it completes its required operations, the Input-Output interrupt handler takes one of these two actions:

   a. It returns control to the interrupted process at the point where the interrupt occurred ("E" in Figure 1-14). To do this, the interrupt handler restores the interrupted environment by resetting the process register values saved in the interrupt's data area.

   b. It passes control to the Dispatcher, another interrupt handler. This typically occurs in the case of significant interrupts where a process of greater priority than the currently executing process has become ready to run as a result of the interrupt. In such cases, the Dispatcher changes the execution environment by selecting the highest-priority process that is ready to run and setting the CPU register values to permit it to run.

Resource Coordination. In a sophisticated operating system where competing processes often request system resources or try to change system tables at the same time, some kind of coordination is an absolute requirement. The Tandem NonStop architecture satisfies this requirement by providing two mechanisms: a counting semaphore facility and a mutual exclusion facility.

Counting Semaphores. The counting semaphore facility permits competing processes to obtain temporary, exclusive access to a particular resource (Figure 1-15). A semaphore (which represents a resource) is composed of a resource count and a waiting list. Processes that take a semaphore decrement the resource count; those that free a semaphore increment the count.

A process actually "takes" or "frees" a semaphore by executing special privileged instructions. When another process tries to take a semaphore that is already taken, the taken semaphore indicates that the resource controlled by the semaphore is currently unavailable (its resource count is exhausted); the requesting process then can be placed on the list of waiting processes and can receive the semaphore when its turn comes. Typically, instructions to take or free semaphores are issued by system procedures called by system processes. Maintenance of the semaphore waiting list, however, is managed by the Dispatcher.

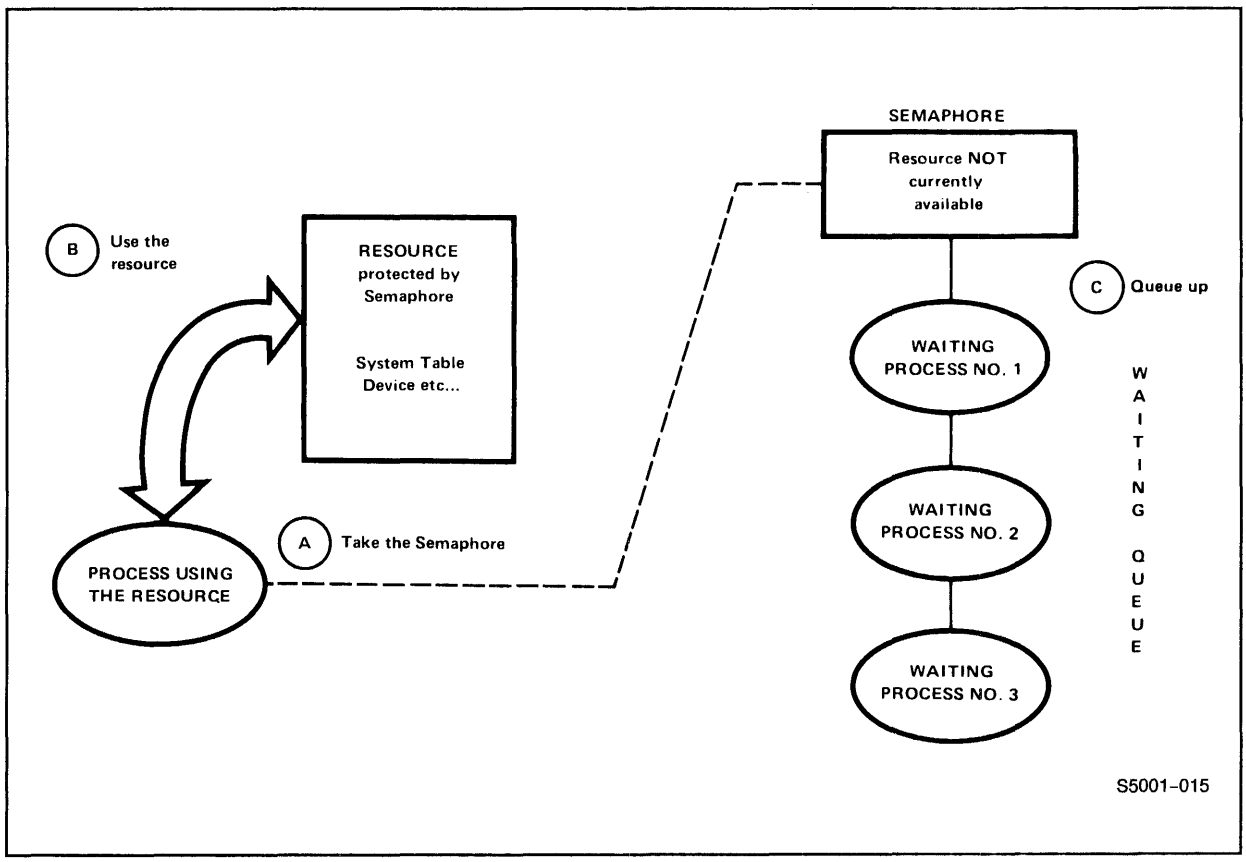


Figure 1-15. Semaphore Use

Mutual Exclusion. A semaphore, as noted in the preceding paragraphs, is intended to protect access to a specific resource. It does not affect other processes not needing the resource related to the semaphore. In certain cases, however, a process needs to extend its exclusive access far beyond the level provided by semaphores. It needs, in fact, to gain absolute control of the entire machine. A process can achieve this level of control through "mutual exclusion." In achieving this control, though, the process must operate in a state where most types of interrupts are temporarily disabled. This effectively trades the majority of the operating system's primary functions for exclusive access to the machine; for instance:

- Because the Dispatcher Interrupt is off, no other process can run.
- Because the memory manager process cannot be dispatched, page faults cannot be processed.
- Because the Time List interrupt is off, timer interrupts are delayed.
- Because the Input-Output interrupts are off, no input-output can be completed.
- Because the Bus Receive interrupts are off, no interprocessor bus transfers can be completed.

To ensure that these functions are disabled only for short periods of time, a process uses mutual exclusion only during very critical operations when no interference can be tolerated. Thus, the process typically disables the interrupts, performs the critical functions, and then immediately reenables the interrupts. First, the process executes the MXON privileged instruction which performs the following:

1. It ensures that the code and data pages required while the interrupts are disabled are present in main memory. The instruction uses two specified ranges, one for code and one for data. It generates dummy memory references in each range to cause page-fault interrupts that bring in the required pages. When a page fault occurs, the instruction is automatically re-executed. This reexecution continues until all required pages are present.
2. Once all required pages are present, MXON saves the old interrupt mask and then disables all interrupts (except the power-fail and high-priority input-output interrupts) by setting the mask bits to 0. This allows the process to secure and retain access to the machine.

After the process executes the critical code, it then executes the MXFF privileged instruction. This instruction restores the old interrupt mask, once again enabling the interrupts.

Interprocess Message Transfers. Since all Tandem NonStop systems have multiple CPUs, and thus multiple operating systems, the GUARDIAN operating systems in all CPUs function together as a group of cooperating processes. These processes communicate by exchanging messages (Figures 1-16 and 1-17) through a "message system." This system consists of a few privileged procedures and a bus interrupt handler, all located in the system library. Message system functions can be called directly by system processes but not by user-written processes (which must invoke the message system implicitly). For example, when a user-written process calls the system procedure named NEWPROCESS, this procedure in turn calls the message system to send a message requesting process creation to the appropriate monitor process.

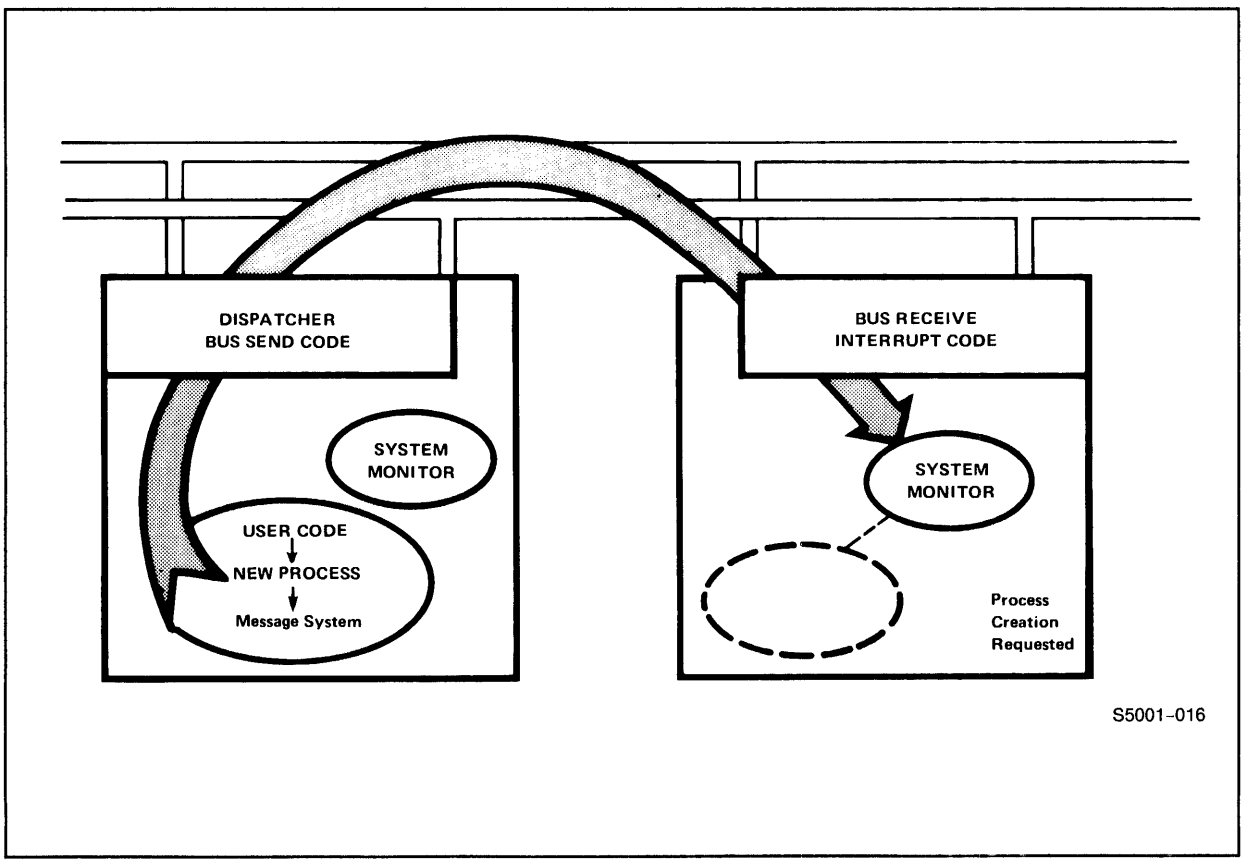


Figure 1-16. Message Transfer Between CPUs

Implicit use of the message system is always accomplished through user-callable system procedures; for instance, in the above example, the call to the monitor was handled by the file system and remained hidden from the user. Users' processes can, however, send messages to other processes directly by opening these other processes as files and writing data to them; this data, of course, is transmitted through the file system. With either explicit or implicit calls, all message system procedures except interrupt handlers run on the calling process's data area and appear to be part of that process.

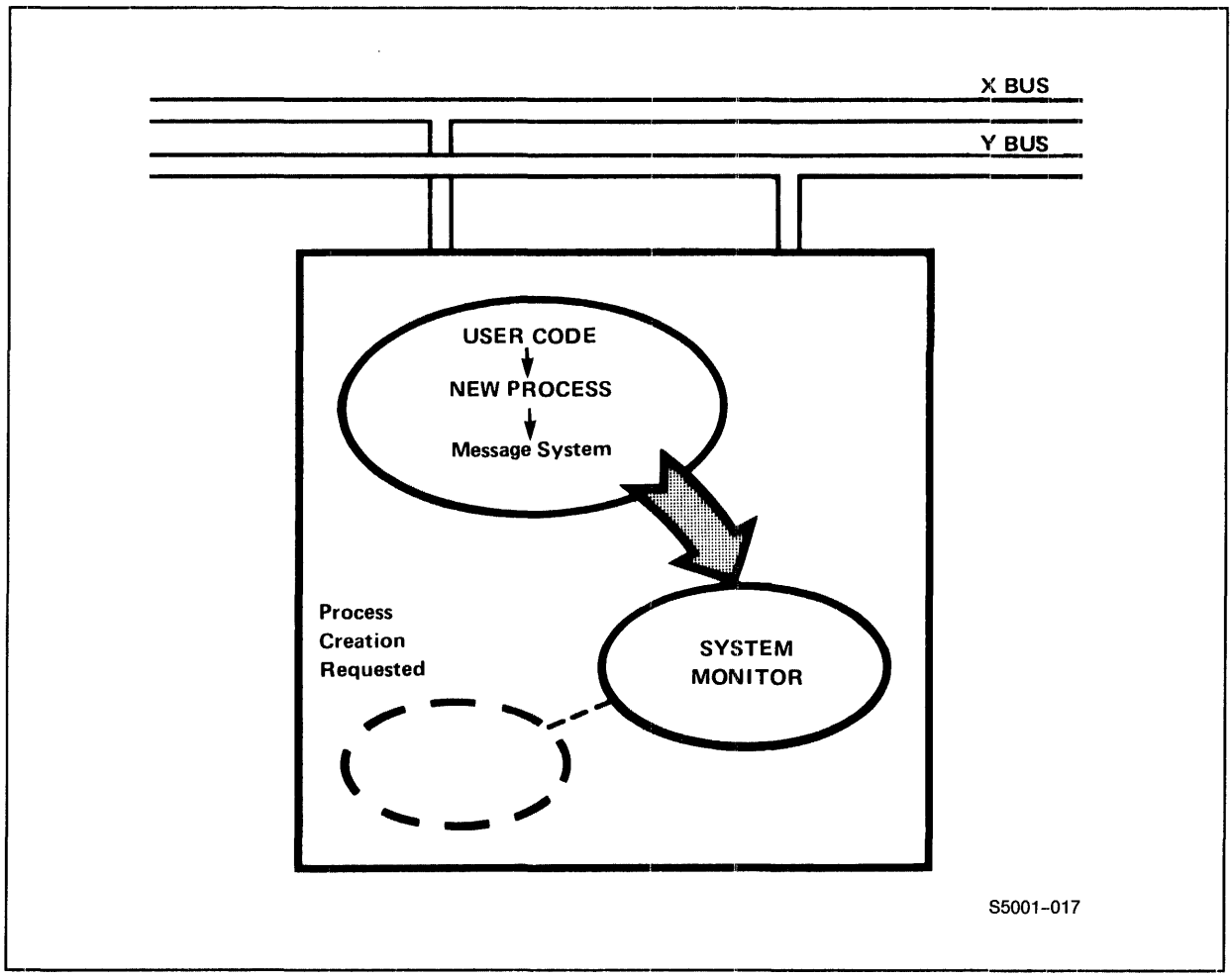


Figure 1-17. Message Transfer Within a CPU

System Data Structures

Several segments of memory (64K words each) are allocated to contain various system data structures, mostly used by kernel procedures and system processes. One of these segments is the system data segment (Figure 1-18). Essentially, as shown, this segment is divided into four major parts: globals, fixed-length tables, variable-length tables, and system pool space (SYSPOOL).

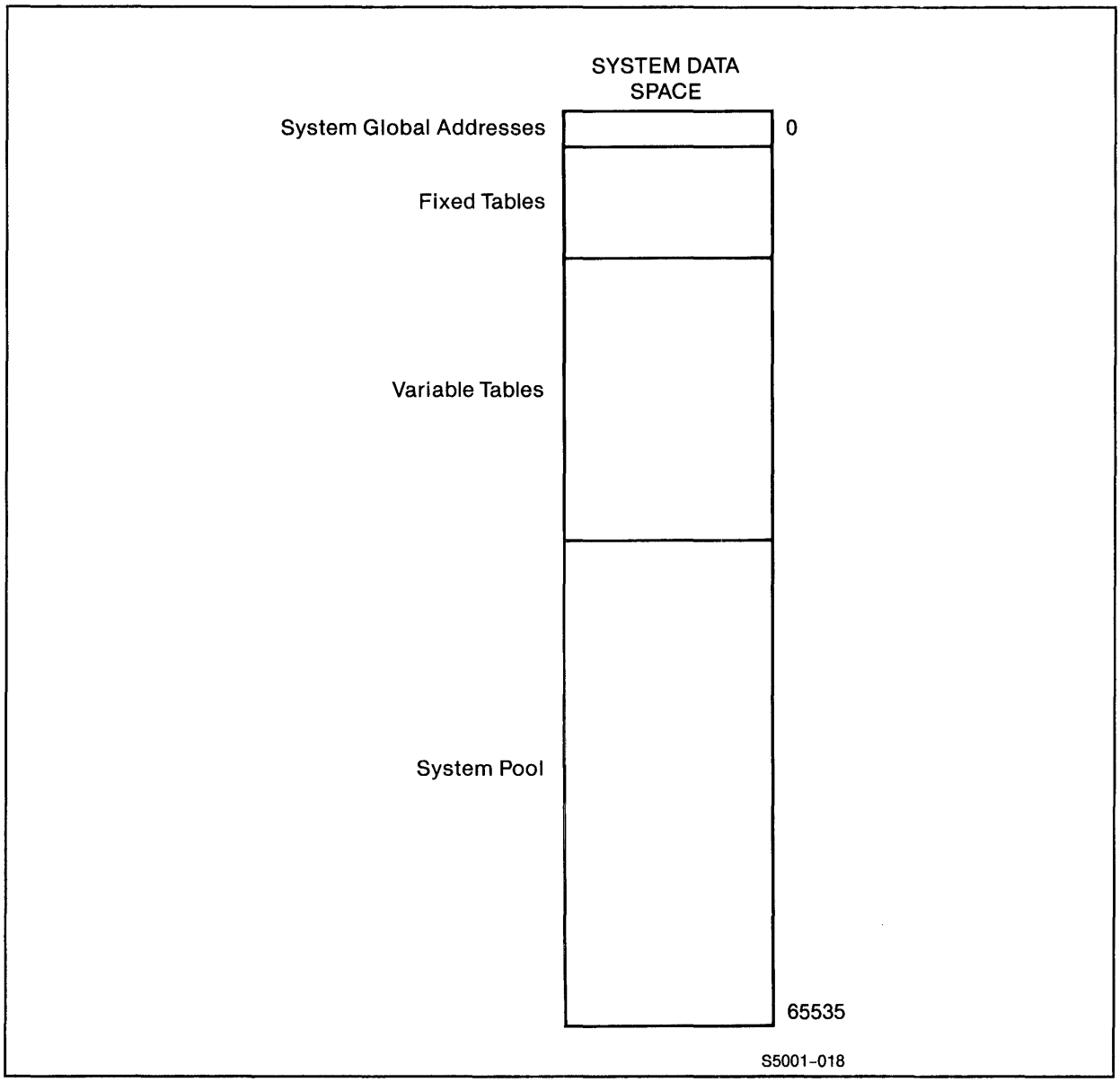


Figure 1-18. System Data Segment

The globals are primarily known-address pointers into the remainder of the system data segment and to system tables in extended memory, thus permitting both microcode and software to reference the primary system tables. The fixed-length tables include, among other things, the Input-Output Control (IOC) table, the Bus Receive Table (BRT), the System Interrupt Vector (SIV), the Subchannel Table (SCT), and the interrupt stacks. The variable-length tables include, among other things, the Controller Table (CTL), Link Control Blocks (LCBs), Process Control Block (PCB) table, and various message system elements. The system pool area is used by the operating system to allocate storage space for various pools, as needed, after which such space is returned (deallocated).

All of the system data segment always remains in CPU memory. Other tables maintained by the kernel are kept in "extended system data segments" and usually are in CPU memory--although unused areas in some cases may not always be. Tables located in extended data segments include the Process Control Block Extension (PCBX), Destination Control Table (DCT), XRAY counters, Network Routing Table (NRT), System Entry Points (SEP), and System Status Messages.

SECTION 2

HARDWARE PRINCIPLES OF OPERATION

This section describes the fundamental operations of the Tandem NonStop system hardware. Also included are a description of the hardware modules, and the operation of data stacks.

FUNDAMENTAL OPERATIONS

To show how the NonStop system provides the means for creating a fault-tolerant application, the following example is given. The example is illustrated in Figures 2-1 and 2-2.

The application consists of a primary application process running in processor module 0 (the primary process is designated A) and its backup process running in processor module 1 (the backup process is designated A'). The coded instructions for A and A' are identical. With the aid of the GUARDIAN software, each can determine whether it is the primary or the backup process, then perform its proper role.

The primary process, while operable, performs all of the application's work. At critical points during each transaction cycle (such as prior to altering the contents of a disc file), the primary process sends a message to its backup process. These messages contain checkpointing information (such as an updated disc record) and keep the backup process up to date on the state of the application. All such messages are the result of checkpointing code that the programmer inserts in the application programs.

The backup process's responsibility, while the primary is operable, is to accept and process the checkpointing messages and be ready to take over the application if the primary process becomes inoperable.

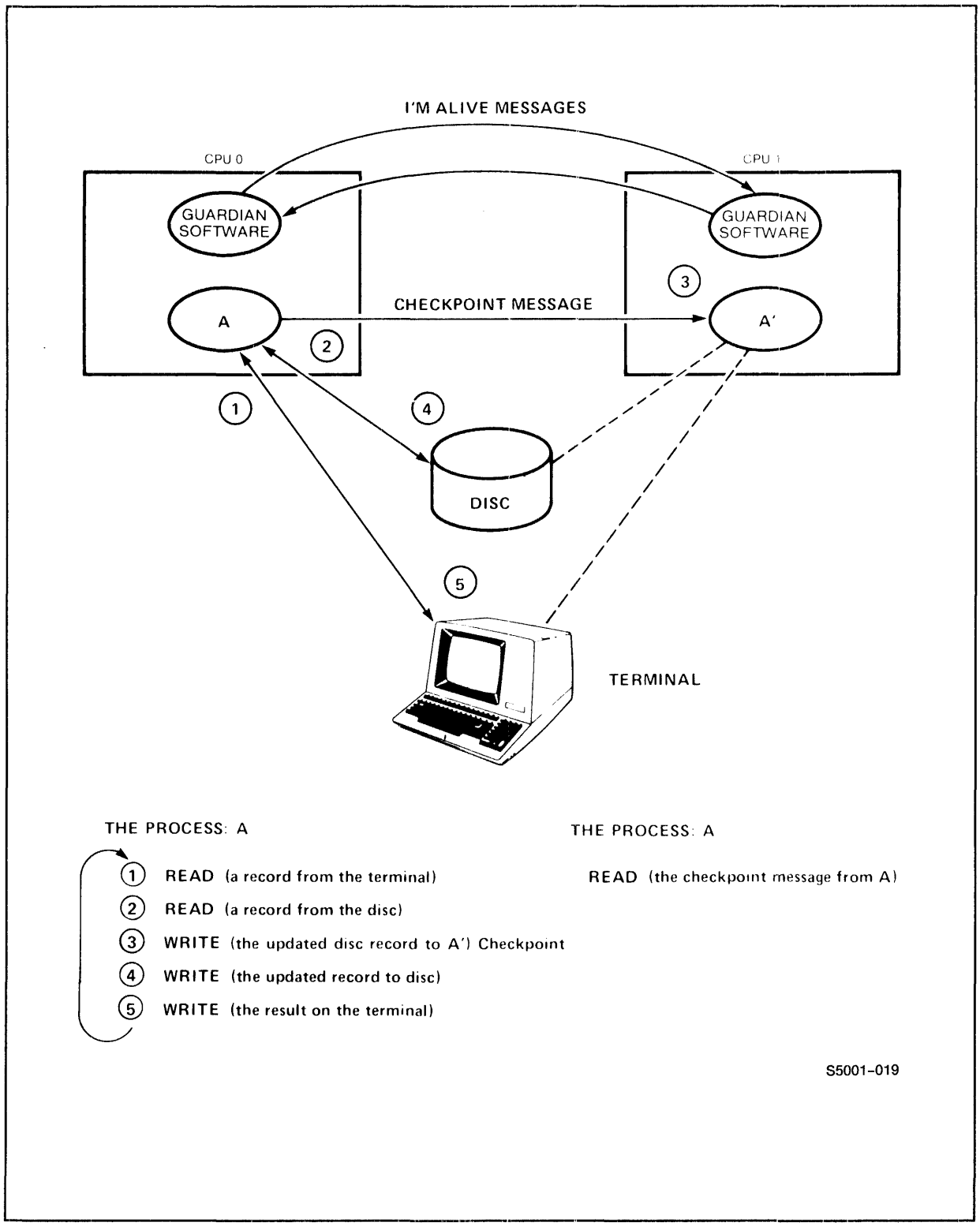
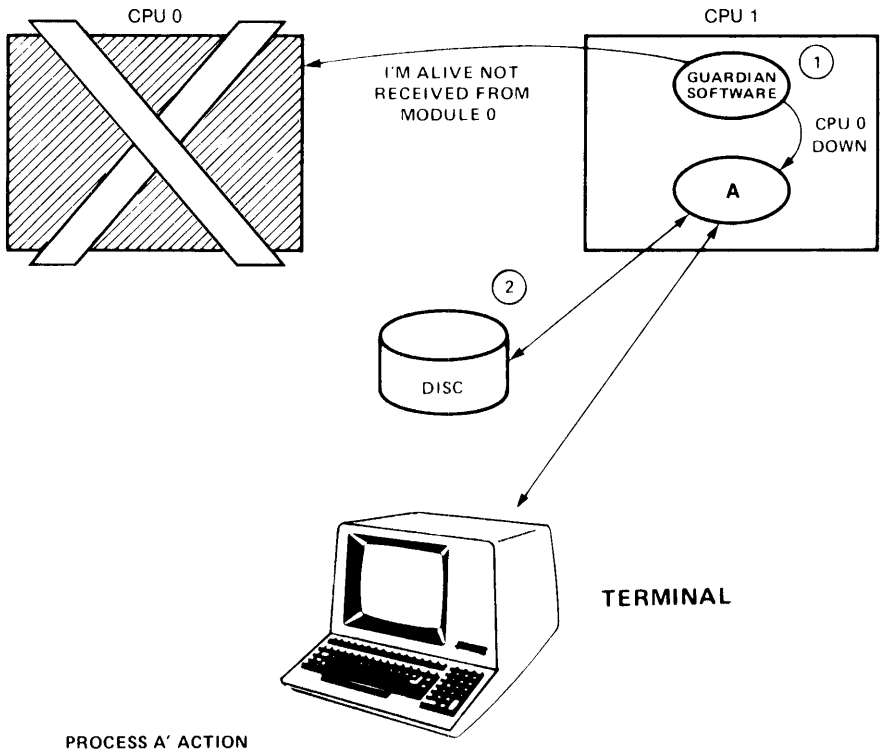


Figure 2-1. Fault-Tolerant Application



PROCESS A' ACTION

(1) READ (the cpu 0 down message)

(2) WRITE (to the disc using the last checkpoint message to ensure update of the record)

Then continue with the same program as A.

READ (a record from the terminal)
READ (a record from the disc)
.
.
.
.

Except that there is no backup for A' at this time, so no checkpoint message is sent.

S5001-020

Figure 2-2. Application Takeover by Backup

If processor module 0 fails (see Figure 2-2), the GUARDIAN operating system in processor module 1 sends a "CPU 0 down" message to backup process A'. This is the signal for the backup process to take over the application's work. First, the backup process uses the latest checkpointing message (e.g., an updated disc record) to complete the transaction that the primary started just prior to its failure, leaving the application's data in the same state as if the primary had completed its last transaction successfully. At that point, the backup becomes the primary and continues with the application's work. (Note that there is no backup process at this time; therefore, no checkpointing messages are sent).

When processor module 0 is reloaded, the GUARDIAN operating system sends a "CPU 0 Up" message to the current primary process (formerly the backup process). The primary process can then start a new backup process running in processor module 0. The primary also begins sending checkpointing information to the backup process. The application is now fully fault-tolerant once again.

PROCESSOR MODULE ORGANIZATION

Instruction Processing Unit

The instruction processing unit (IPU) has four functions: 1) to execute machine instructions, 2) to provide for the orderly interruption of a running process, 3) to map logical to physical memory, and 4) to transfer data from the interprocessor buses into memory (this is invisible to the executing process and is handled entirely by the IPU's microprocessor).

A program's instructions reside in memory. In order to execute an instruction, it is first fetched from a location in memory determined by the address held in an IPU register. The instruction is loaded into another IPU register and is decoded by the hardware to determine what sequence of microinstructions must be used to execute the instruction. During execution of the instruction, one or more memory transfers can occur, the IPU's scratchpad registers can be used to hold intermediate computations, and operands can be added to or deleted from the IPU's Register Stack.

The IPU is "pipelined," processing multiple instructions at once. For example, while the current instruction is being executed, the next instruction in sequence can be fetched from memory at the same time.

For a NonStop II processor, the microinstruction cycle time is 100 nanoseconds; microinstructions are 32 bits (plus parity) in length. For a NonStop TXP processor, the microinstruction cycle time is 83 nanoseconds; microinstructions are 109 bits long (74 bits for horizontal control store and 35 bits for vertical control store) plus parity.

An IPU's basic instruction set consists of approximately 235 instructions. These include arithmetic operations such as add, subtract, multiply, and divide; logical operations such as AND, OR, and exclusive OR; bit shift and deposit; block (multiple-element) moves, compares,and scans; procedure call and exit; interprocessor bus send; and the input-output instructions. All instructions are 16 bits in length.

Processor modules equipped with the decimal arithmetic option have an additional 14 instructions (6 decimal arithmetic instructions are standard in all processors). These instructions operate on four-word operands and perform operations such as add, subtract, multiply, divide, negate, compare, and round. (See Decimal Arithmetic Option headings in Section 9, "Instruction Set".) Modules equipped with the Floating-Point option have an additional 41 instructions for doubleword and quadrupleword (extended) floating-point arithmetic and related operations. (See "Floating-Point Arithmetic" and "Extended Floating-Point Arithmetic" in Section 9.) With these options, a processor has a total of approximately 290 instructions.

Two modes of process execution are provided: privileged and nonprivileged. A process executing in nonprivileged mode is not permitted to execute the instructions designated as privileged. Privileged instructions are associated with operations that, if performed incorrectly or inadvertently, could have an adverse effect on other processes or the operating system. These privileged operations include: interprocessor bus send, input-output, changes to map registers, execution of privileged procedures, and access to system data. Normally, only the GUARDIAN operating system executes in privileged mode; application (user) processes execute in nonprivileged mode. Privileged operations are performed for nonprivileged processes through calls to operating system procedures. An attempt by a nonprivileged process to execute a privileged instruction causes the process to be trapped (interrupted).

The interrupt function provides for the orderly transfer of IPU control from an executing process to one of several routines in the operating system called interrupt handlers. This transfer of control is called an interrupt. Interrupts occur for several reasons. Among them are: data received over the interprocessor bus, completion of an I/O transfer, memory error, memory page absent, instruction failure (e.g., attempt by a nonprivileged process to execute a privileged instruction), and power failure.

Memory

Physical memory is the storage space provided by the actual solid-state memory locations available to a processor on its memory boards. There can be up to four memory boards. Thus, for example, if a processor module contains four memory boards having 2 megabytes per board, that processor's physical memory is 8 megabytes. The maximum addressing range for physical memory is 16 megabytes, and this defines the maximum physical memory size. Since each processor module contains its own memory boards, physical memory is private to the processor.

Data is stored in physical memory in the form of 16-bit words; 1024 words comprise a page. Although access to physical memory is by word on word boundaries, specific instructions also provide element access to bytes, doublewords, and quadruplewords. The NonStop TXP processor can access in parallel up to four words (64 bits) on a selected memory board.

Logical memory is memory as perceived by a particular process, being some subset of the total virtual memory space. (The addressing range for virtual memory in a single processor is one gigabyte; virtual memory consists of all "segments" in this range that are currently allocated.) The logical memory for any given process is defined as a certain number of virtual memory segments, and is independent of the processor's physical memory.

Memory addressing can be defined in terms of logical addresses or physical addresses.

A logical address most commonly consists of 16 bits; a 16-bit address is capable of addressing a maximum of 64K words (i.e., one segment of memory). A short address is a 16-bit address plus three bits to specify one of six short address spaces. Short addresses and short address spaces are described later under "Memory Access" in Section 5.

Because a process consists of three independently addressable areas (one or two code spaces and one standard data segment), and has access to the system code spaces, and because code spaces can consist of multiple code segments, a single process potentially can access over 4 megawords (32 user code and library segments, 33 system code and library segments, and one data segment) without using extended addressing. Extended addressing (32-bit addresses) permits an even greater range of logical memory access, and is described later under "Extended Addressing" in Section 5.

A physical address consists of 23 bits. A 23-bit address provides an addressing range of sixteen megabytes, thus it is capable of referencing any location in physical memory.

Many application processes and parts of the operating system can reside in physical memory concurrently. As each process is granted execution time in the processor, its logical memory space becomes part of the currently accessible portion of physical memory--that is, the process's segments become "mapped."

Mapping converts logical addresses to physical addresses; i.e., mapping makes physical pages scattered through memory appear to the program to be a contiguous block of memory.

In a NonStop II processor, address translation for the short address spaces is implemented through hardware map registers. (Each CPU actually has sixteen hardware maps.) Each map consists of 64 entries (registers), and each entry points to an individual physical page of memory. Thus, a map is capable of defining one segment of logical memory. The sixteen hardware maps are described later under the heading "Memory Access (NonStop II Processor)" in Section 5.

In a NonStop TXP processor, address translation for the short address spaces is provided within a larger (2048-entry) hardware register array called "PCACHE". This PCACHE permanently maps some of the short address spaces (system code and data, for example), but primarily functions as a cache of page mappings--one page at a time, as needed, rather than all pages of a given segment. PCACHE is described later under the heading "Memory Access (NonStop TXP Processor)" in Section 5.

The data path between memory and other processor module functions is 16 bits wide. All data is verified for accuracy when it is read from memory. Six error correction bits are appended to each 16-bit word when it is stored. The use of the six error correction bits in the semiconductor memory permits the hardware to correct all single-bit errors automatically and to detect all double-bit errors. The detection of a memory error (whether correctable or uncorrectable) causes an interrupt to an operating system interrupt handler, which takes appropriate action.

Input-Output Channel

Each processor module has its own I/O channel that is capable of transferring data between I/O devices and memory at full memory speed. I/O operations, which are controlled by the operating system, are initiated by setting up an entry in a table in memory and then executing an Execute I/O (EIO) instruction. Once initiated, data transfer occurs concurrently with software process execution. When the I/O operation completes, the currently executing process is interrupted, and control of the IPU is transferred to an operating system interrupt handler.

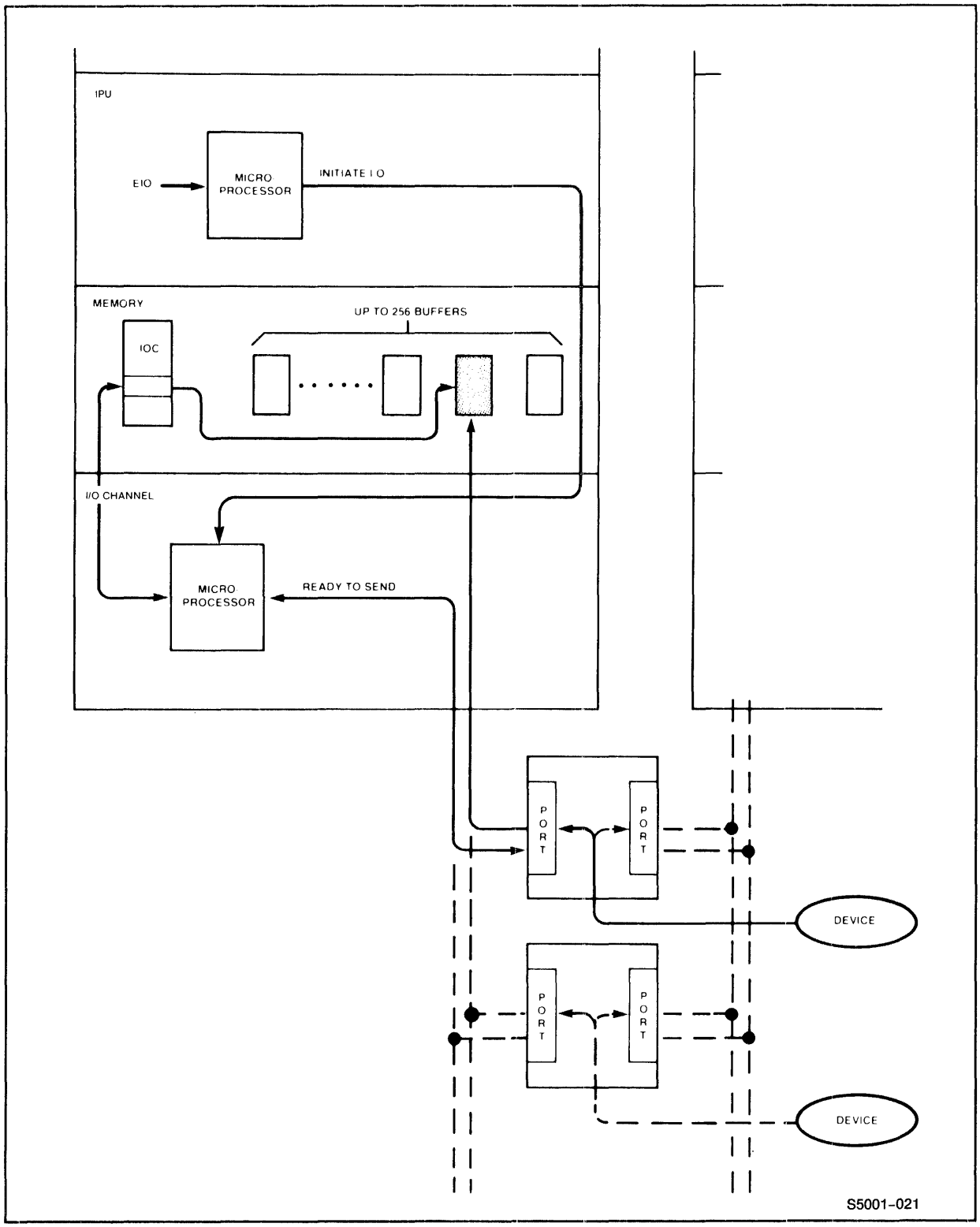


Figure 2-3. Input-Output Channel

Each channel is capable of addressing 256 I/O devices, addressing each as a separate "subchannel." A single I/O operation is capable of transferring data in blocks of from 1 to 65,535 bytes.

The table to control I/O transfers is called the I/O Control (IOC) table. Each processor module has its own IOC table. (See Figure 2-3.) The IOC table is maintained both by the operating system and be microcode. The IOC table contains up to 256 entries, corresponding to the 256 possible devices (subchannels) on that processor's channel; each entry contains a buffer address (in one of the I/O buffer segments) and a count of the number of bytes to be transferred. The use of the IOC table permits an I/O channel to run any number of devices (up to 256) concurrently while maintaining control on a device-by-device basis. When the number of bytes indicated in the IOC have been transferred, the device interrupts the currently executing process.

Data is buffered by each controller so that data is transferred in bursts through the channel at memory speed (the number of bytes in a "burst" depends upon the type of controller). Controllers are designed so that they signal the channel prior to actually emptying their buffers (during a write operation) or filling their buffers (during a read operation). This gives the channel ample time to respond, thereby providing a means to avoid data overrun. All 256 devices can be transferring simultaneously, with bursts from one device being interleaved with bursts from others, subject to I/O data rate configuration limits.

Interprocessor Bus Interface

The NonStop system has two interprocessor buses (see Figure 2-4). Each bus functions independently of the other, transferring data from one processor module's memory to another processor module's memory. Both buses can be in use simultaneously.

Data is transferred over each interprocessor bus at a maximum rate of 13.33 megabytes per second. Each bus is capable of transferring data among all processor modules concurrently on a packet-multiplexed basis.

An interprocessor bus transfer involves two processor modules: the sender module and the receiver module. The transfer is initiated by the sender when a SEND instruction is executed. The receiver module checks the incoming packet for correct transmission (using checksum, sequence number, and destination and receiver numbers), and directs the incoming data to a main

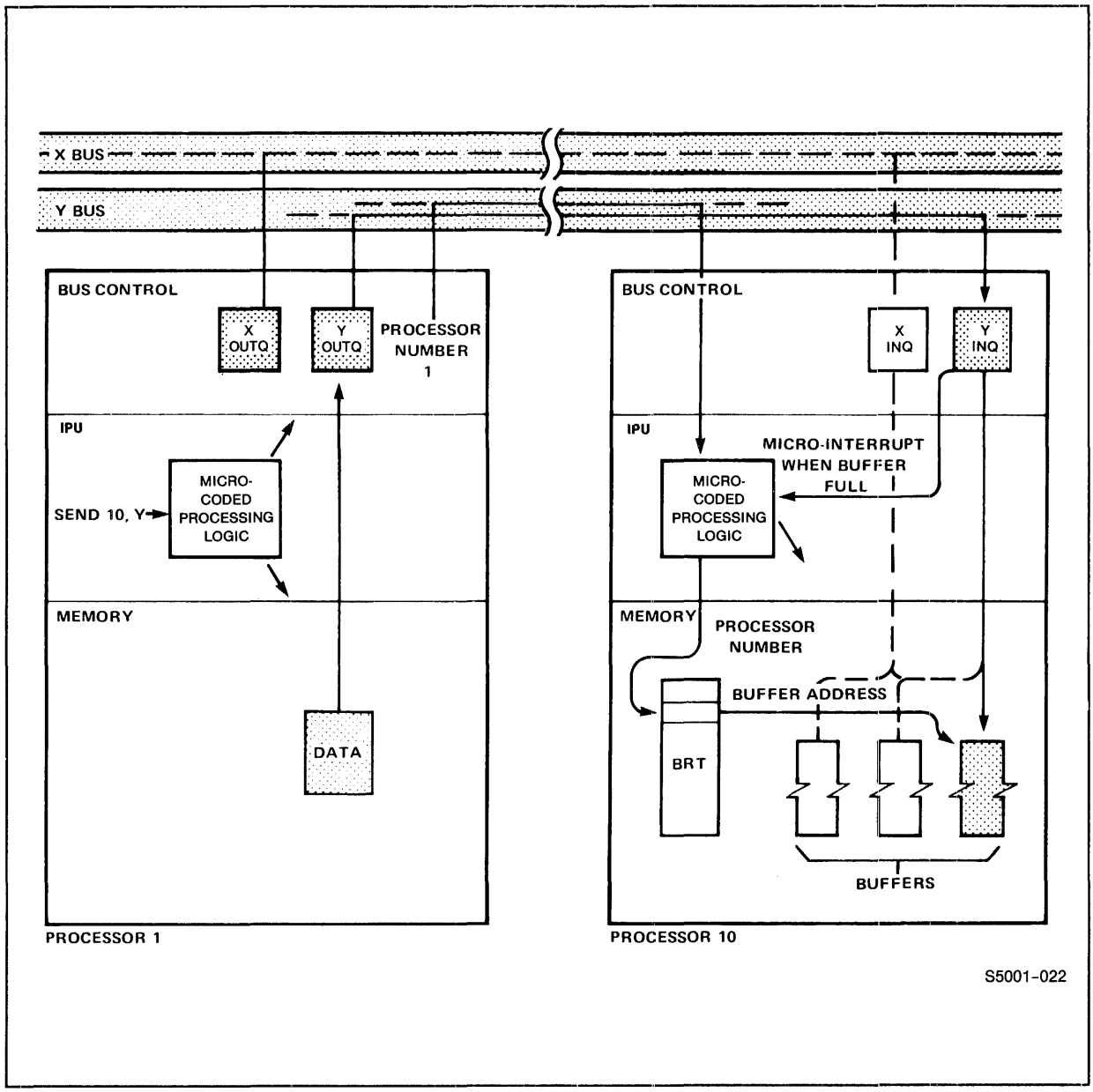


Figure 2-4. Interprocessor Bus Interface (NonStop II Processor)

memory buffer indicated by a firmware-known, software-maintained table--the Bus Receive Table (BRT). This table, the BRT, is then updated.

The SEND instruction can transmit blocks of 1 to 65,535 bytes to a designated processor module over one of the buses. Data is actually sent across a bus in packets of 16 words (a routing word, a sequence word, 13 data words, and a checksum word); each processor module contains two high-speed 16-word buffers (one for each bus) for receiving the incoming information. These buffers are designated INQ X (for the X bus) and INQ Y (for the Y bus). Transfers into the buffers occur simultaneously with IPU microprogram execution; when a buffer fills, the IPU microprogram is interrupted, and a special microroutine moves the contents of the buffer into memory.

Each processor module's main memory contains a BRT. The BRT is known by the firmware and is maintained by the operating system. It is used to direct the incoming bus data to a specified location in a processor module's memory. The BRT contains 16 entries (corresponding to the 16 possible processor modules in a system); each entry specifies an expected packet sequence number, a buffer address where the incoming data is to be stored, and the number of bytes expected. When the expected number of bytes has been received, the currently executing process is interrupted, and the process for which the message is intended is notified.

Other Processor Components

In addition to the four main processor components just described (the IPU, memory, I/O channel, and interprocessor bus interface) each processor in a NonStop system contains several other important components. These are discussed briefly in the following paragraphs. Figure 2-5 illustrates these components, showing their relationships to each other and to the four major components already discussed.

Clock Generator. The clock generator is the main processor clock. It provides the synchronization of all hardware functions within the processor. The NonStop II processor's clock has a full-cycle time of 100 nanoseconds (10MHz) and a half-cycle time of 50 nanoseconds. The NonStop TXP processor clock has a full-cycle time of 83.33 nanoseconds (12MHz) and a half-cycle time of 41.66 nanoseconds.

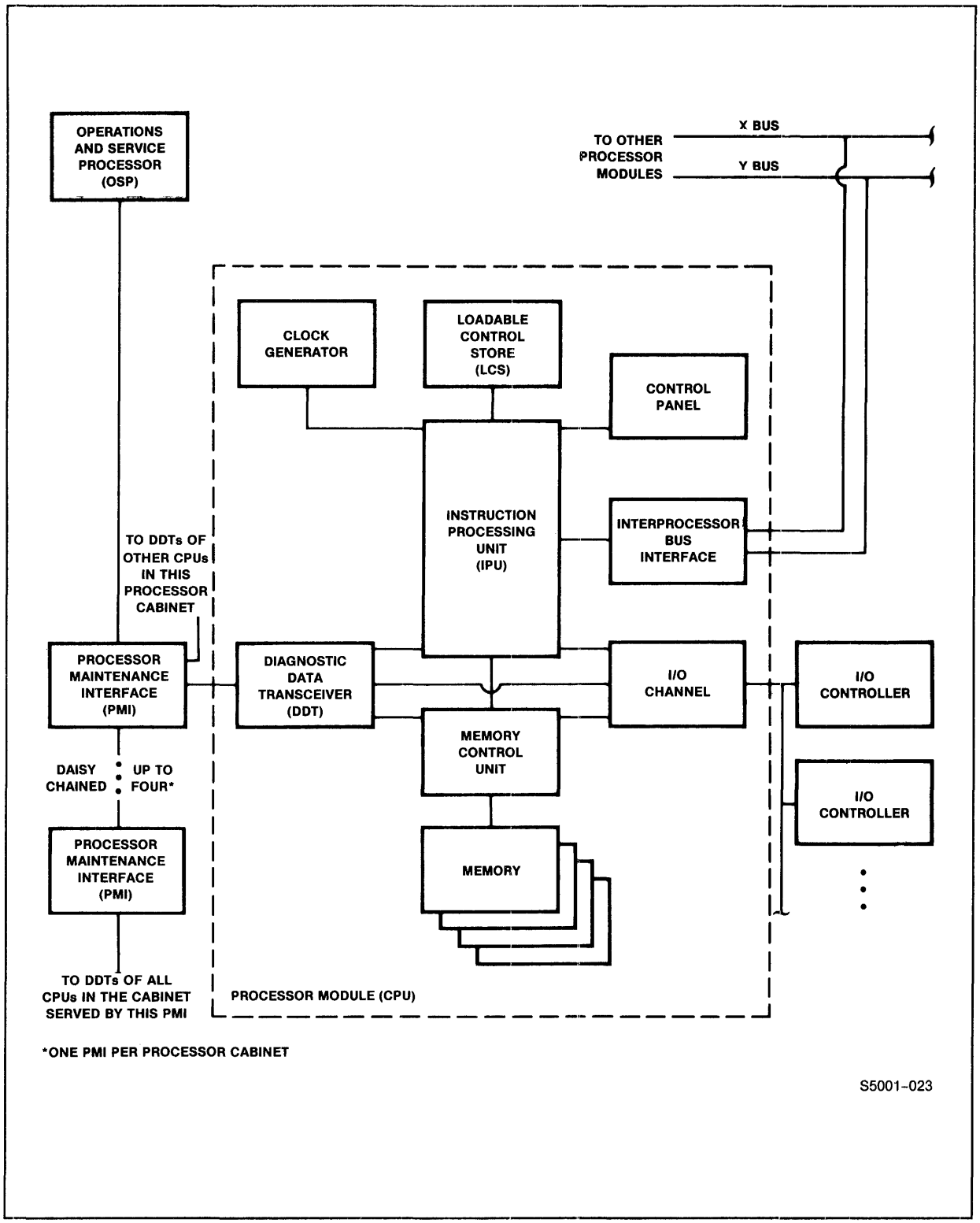


Figure 2-5. Block Diagram of Processor Hardware

Loadable Control Store. The Loadable Control Store (LCS) contains microinstructions for use by the IPU. Each machine instruction causes the IPU to execute a specific set of microinstructions to implement the functions of that machine instruction. The LCS cannot be written to by user programs, but it can be loaded with new versions of the system microcode and microcode options as they are purchased from or supplied by Tandem.

Control Panel. The control panel allows operators and maintenance personnel to interact directly with each processor. The control panel can be used to reset a processor, cold load a processor, ready a processor for reload, and give visual indications of a processor's status. It also can be used to initiate some microdiagnostics.

Memory Control Unit. The memory control unit (MCU) provides access to memory for both the I/O channel and the IPU. The MCU queues memory requests by execution priority; provides overlapped access, mapping of logical to physical memory (NonStop II processor only) error control, and error reporting; and provides semiconductor memory refresh timing capability.

Diagnostic Data Transceiver. One Diagnostic Data Transceiver (DDT) is associated with each processor in the system. Connected to the Operations and Service Processor (OSP) through the Processor Maintenance Interface (PMI), the DDT communicates at two distinct levels, as directed by the microcode in the LCS or by a process running in the CPU. It can accept commands from the OSP to communicate with the operating system and run diagnostics for operations or fault isolation. It can also report the status of the IPU, MCU, I/O channel, and LCS to the OSP.

Processor Maintenance Interface. The Processor Maintenance Interface (PMI) provides a common interface point for up to four processors in a cabinet to communicate with the OSP. If there is more than one processor cabinet in the system, a PMI is added for each cabinet, and the PMIs are connected together.

The PMI has switch functions that regulate communication between processors and between a processor and the OSP. The PMI also has indicator lights showing DDT status. In addition, it provides signal-level conversion; it connects to the processors through differential signals, which it passes on to the OSP. Finally,

the PMI notifies the DDT of the speed at which the local or remote OSP is operating.

OPERATIONS AND SERVICE PROCESSOR

The Operations and Service Processor (OSP) is the control center for the NonStop system. Through the OSP, operators and maintenance personnel can easily and flexibly invoke many low-level system functions, including all the essential functions of the control panel for each processor.

The OSP provides both local and remote operations and maintenance capabilities. As previously described, it is connected to each processor through the PMI and the DDT.

The OSP subsystem is made up of six components:

- Processor--The processor is the central part of the OSP. (The OSP processor is not to be confused with a processor module, or CPU.) Most of the OSP functions are controlled by the processor. The processor provides intelligence and coordination of the OSP.

- Floppy Disc Drives--A floppy disc drive is used to load the OSP operating system and diagnostics from floppy disc (diskette) into the OSP. Two floppy disc drives and associated power supplies are provided for failure tolerance.

- Switches and Indicators--The OSP switches and indicators provide access control and OSP status information.

- OSP Terminal--The OSP terminal provides an easy and flexible operation and maintenance interface with the OSP and the NonStop system. Function keys are provided to allow fast interaction with the OSP.

- Modem--The modem included in the OSP allows communication with remote OSPs and remote terminals. Maintenance can be performed from all of these devices. Operations can be performed from a remote OSP or a remote OSP terminal.

- Optional Hard-Copy Printer--Optional printers are available for hard-copy logging of system console activity.

HOW THE HARDWARE EXECUTES PROGRAMS

Code and Data Separation

Programs executing as processes in a CPU's memory are physically separated into two areas: code segments containing machine instructions and program constants, and data segments containing program variables. See Figure 2-6. The code segments of a process can be thought of as read-only storage, since no machine instructions can write into them. Since code segments cannot be modified, they can be shared by a number of processes.

Procedures

Programs are functionally separated into blocks of machine instructions called procedures. A procedure, like a program, has its own local data area (in the process's data segment). A procedure (that is, the block of instructions that a procedure represents) is called into execution when a procedure call instruction (PCAL, XCAL, or DPCL) is executed. The call instruction saves the caller's environment and transfers control to the entry-point instruction of the procedure.

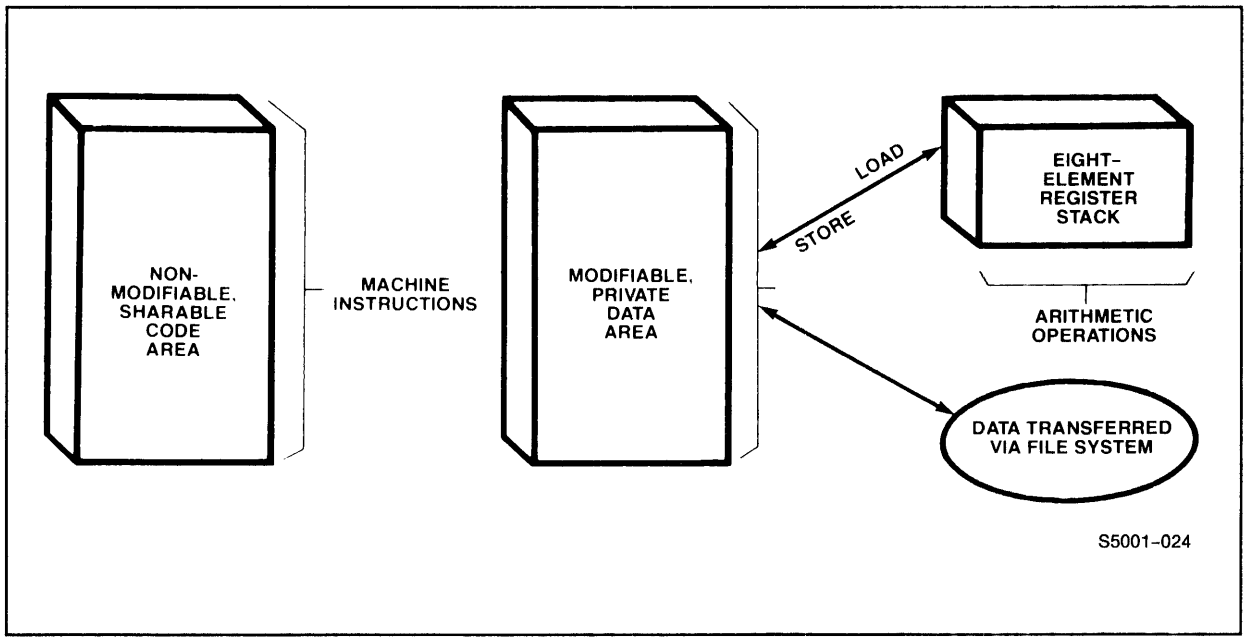


Figure 2-6. Code and Data Separation

The procedure's instructions are then executed. The last instruction that a procedure executes is an EXIT instruction. The EXIT instruction restores the caller's environment and transfers control back to the caller's next instruction.

A procedure, while it executes, has its own local data area. This area is allocated for a procedure each time the procedure is called and is deallocated when the procedure exits (see "Memory Stack"). The procedure can also access a shared global data area, which is accessible to all procedures of the process. The global data area and all the memory used for procedure local data areas are contained in the process's data segment.

Procedures can be written so that they can receive parameter information (arguments), perform computations using the parameters, then return results to the caller. (The machine instructions for passing parameters and returning results are generated automatically by compilers.)

Procedures that are outside the currently executing code segment (that is, in some other code segment accessible to this process) are accessed by means of an "external call." For example, operating system functions (such as file system operations) are performed by calling procedures that are in one of the system library segments. An external procedure is called when an External Procedure Call (XCAL) instruction is executed. This is discussed later in Section 4 under "Calling External Procedures."

Memory Stack

The first half of a process data segment is organized in memory as a "stack." A stack is a storage allocation method in which the last item (or block of items) added is the first item removed--like a stack of dishes. The local areas for procedures are blocks of data items in the memory stack. A procedure's local data is allocated in the memory stack only while it executes; after a procedure returns to the point where it was called, its data area is deallocated and can be used by another procedure called later. Therefore, the total amount of memory space required by a program is kept to a minimum.

Figure 2-7 illustrates the memory stack manipulations ("Data Area") during a sequence of procedure calls ("Code Area"). Sequence number (1) shows the memory stack when procedure A starts executing. At (2), a call to procedure C pushes C's parameters onto the stack (3), along with the link back to A. At (4), C begins to execute, using the stack for its local variables (5). Then a call to B (6, 7, 8) pushes B's parameters onto the

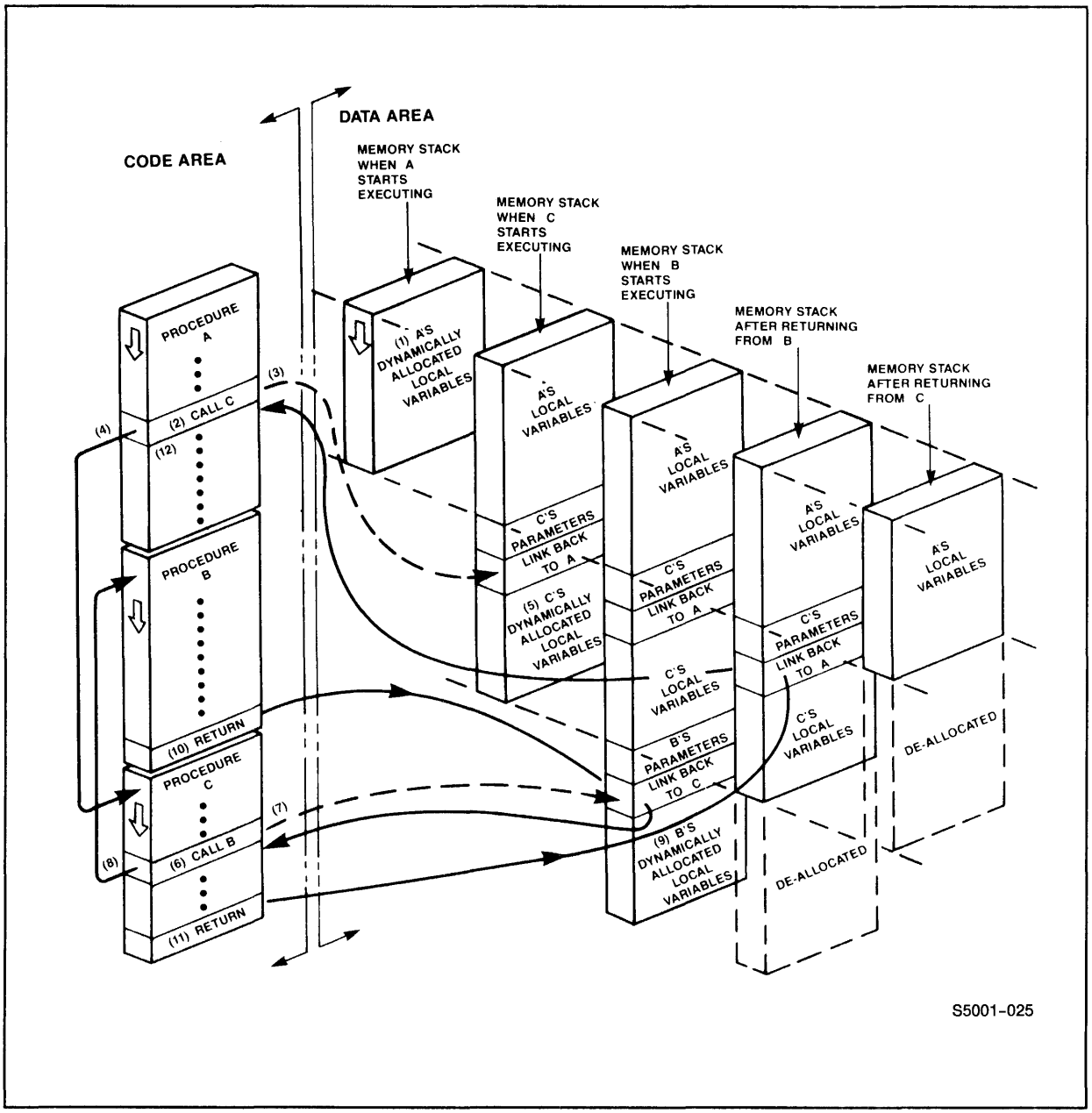


Figure 2-7. Memory Stack Operation

stack, along with the link back to C, and B uses the stack for its local variables (9). Then, when B completes, it executes a return (10) back to C, deallocating its local variables, calling parameters, and return link from the stack. Procedure C, in turn, runs to completion and executes a return (11) back to A, deallocating its unneeded information from the stack. Procedure

A continues its execution (12), with the stack back to the condition it was in prior to the calls; no unneeded data from these manipulations remains behind to waste memory.

Register Stack

Each instruction processing unit contains a "Register Stack," consisting of eight separate registers. Each register stores one 16-bit word. The Register Stack provides a highly efficient means of executing arithmetic operations; operands are loaded onto the stack, arithmetic operations are performed, the operands are deleted, and a result is left on the stack. An add of two 16-bit numbers is illustrated in Figure 2-8.

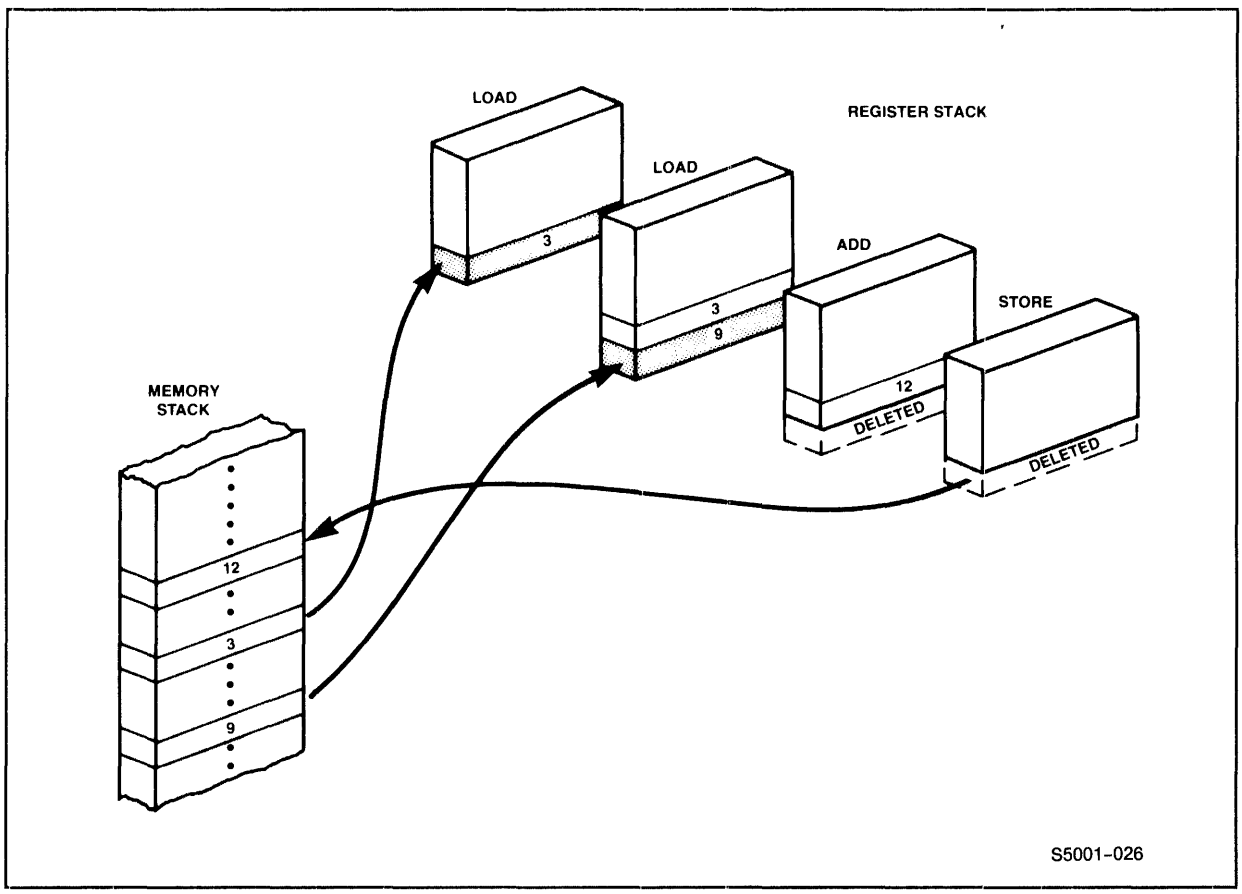


Figure 2-8. Register Stack Operation

The use of the Register Stack is usually transparent to programmers using programming languages, and most application programming does not require explicit stack operations. The language compilers automatically generate the machine instructions for efficiently using the Register Stack.

SECTION 3

DATA FORMATS AND NUMBER REPRESENTATIONS

DATA FORMATS

The basic unit of information in the NonStop II and NonStop TXP processors is the 16-bit word. However, individual access to and operations on single or multiple bits (bit fields) in a word, 8-bit bytes, 16-bit words, 32-bit doublewords, and 64-bit quadruplewords are supported. See Figure 3-1.

In this manual, a number surrounded by brackets is used to denote an individual element (that is, word, doubleword, byte, or quadrupleword) in a block of elements:

```
block [element]
```

For example, to indicate the fourth element in a word block (beginning with element 0), the following notation is used:

```
WORD [3]
```

When referring to a block of words (or any elements), the first element is indicated by the element number that is the lowest numerically; the last element has the highest element number. The following notation is used to denote a block of elements:

```
block [first element:last element]
```

For example, to indicate the second through twentieth words in a block, the following notation is used:

```
WORD [1:19]
```

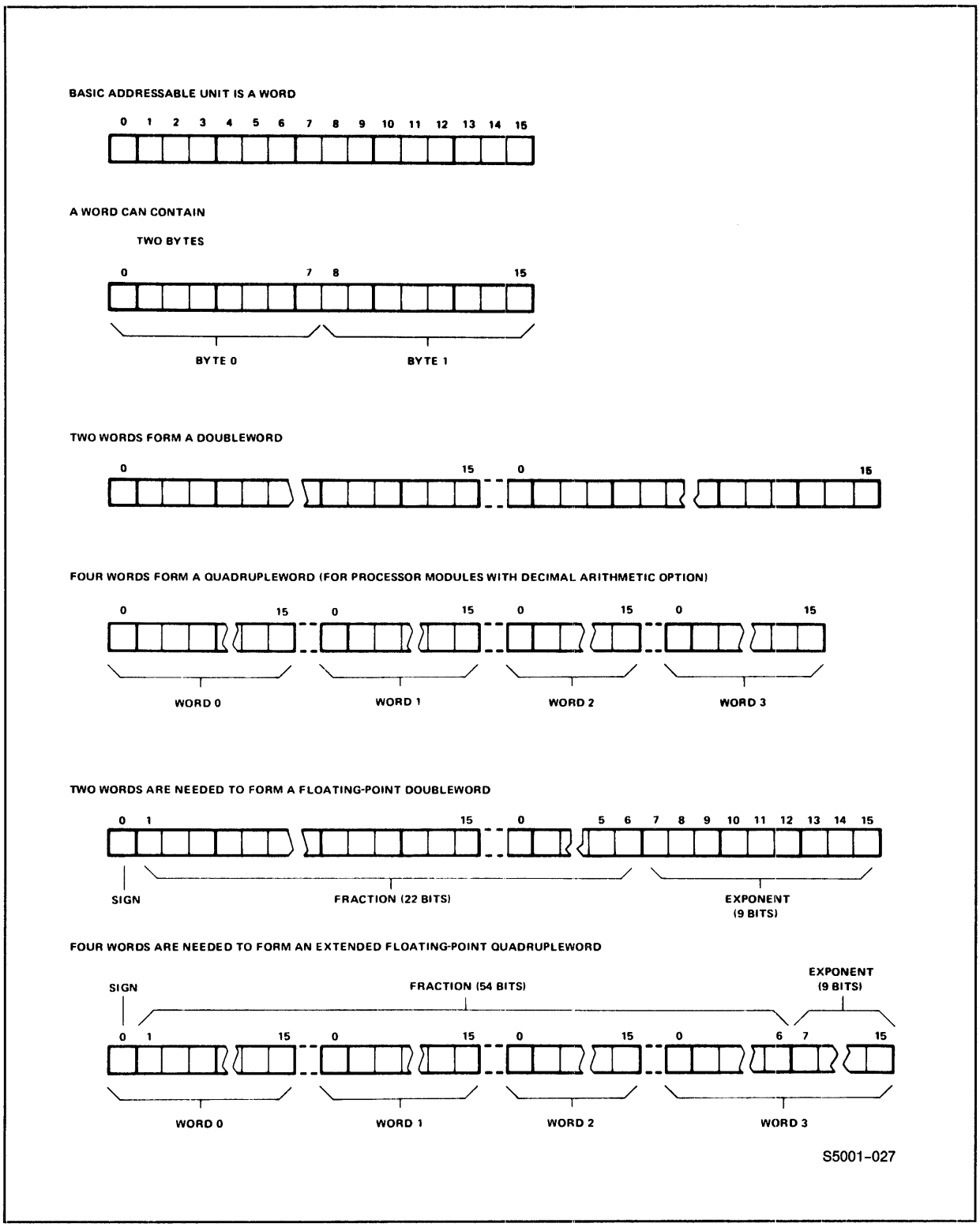


Figure 3-1. Data Formats

Words

The 16-bit word defines the machine instruction length and logical addressing range for the NonStop II and NonStop TXP processors. The 16-bit word is the basic addressable unit stored in memory. The first word in each segment (i.e., code or data) of logical memory is addressed as WORD[0]; the last addressable location is WORD[65535]. This is shown in Figure 3-2.

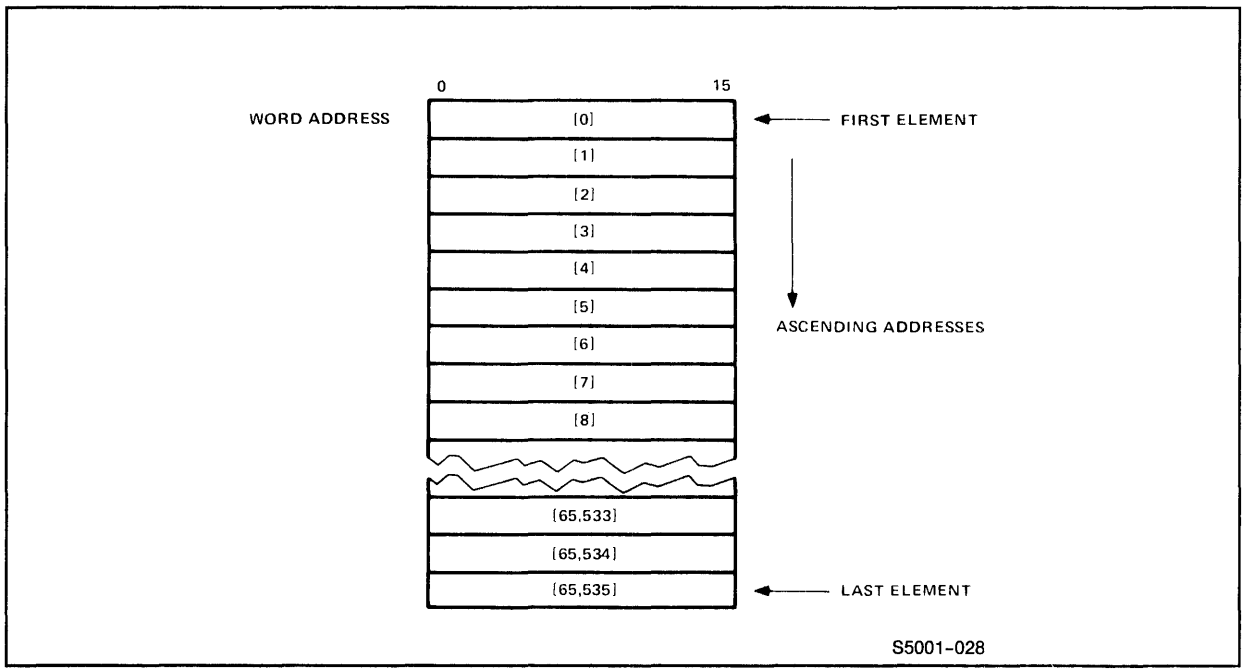


Figure 3-2. Word Addressing

The following instructions are provided for referencing words in logical memory:

LOAD	Load word into Register Stack from data segment
STOR	Store word from Register Stack into data segment
LWP	Load Word into Register Stack from Program (code seg)
NSTO	Nondestructive Store word from Register Stack into data segment
ADM	Add word from Register Stack to word in Memory (data segment)
LDX	Load Index Register from data segment
LWA	Load Word via A
SWA	Store Word via A

LWX	Load Word Extended
LWXX	Load Word Extended, Indexed
SWX	Store Word Extended
SWXX	Store Word Extended, Indexed
ANG	AND to Current Data
ORG	OR to Current Data
ANX	AND to Extended Memory
ORX	OR to Extended Memory
LWUC	Load Word from User Code Segment

Two instructions operate on blocks of words:

MOVW	Move Words from one memory location to another
COMW	Compare Words in one memory location with another

Bits

The individual bits in a word are numbered from zero (0) through fifteen (15), from left to right:

```
                              1 1 1 1 1 1
WORD: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
```

The following notation is used in this manual (and in the TAL language) to describe bit fields:

```
WORD.<left bit:right bit>
```

For example, to indicate a field starting with bit 4 and extending through bit 15, the following notation is used:

```
WORD.<4:15>
```

To indicate just bit 0, the following is used:

```
WORD.<0>
```

Bytes

Two bytes can be stored in a 16-bit word. The most significant byte in a word occupies WORD.<0:7> (left half); the least significant byte occupies WORD.<8:15>. The 16-bit address provides for element addressing of 65,536 bytes.

In the data segment, byte-addressable locations start at BYTE[0] and extend through BYTE[65535]. Two bytes are stored in each

word; therefore the first 32,768 words of the data segment (WORD[0:32767]) can store 65,536 bytes. The upper half of the data segment, WORD[32768:65535], is not byte addressable without the use of extended addressing.

In the code segment, byte addresses are computed by the hardware relative to whether the current setting of the P (for Program counter) Register is in the lower or the upper half of the code segment. Therefore, the entire code segment (WORD[0:65535]) is byte addressable, as explained in the description of the LBP instruction in Section 9.

The IPU converts a byte address to a word address and bit field in that word, as shown in Figure 3-3. In other words, bit 15 of

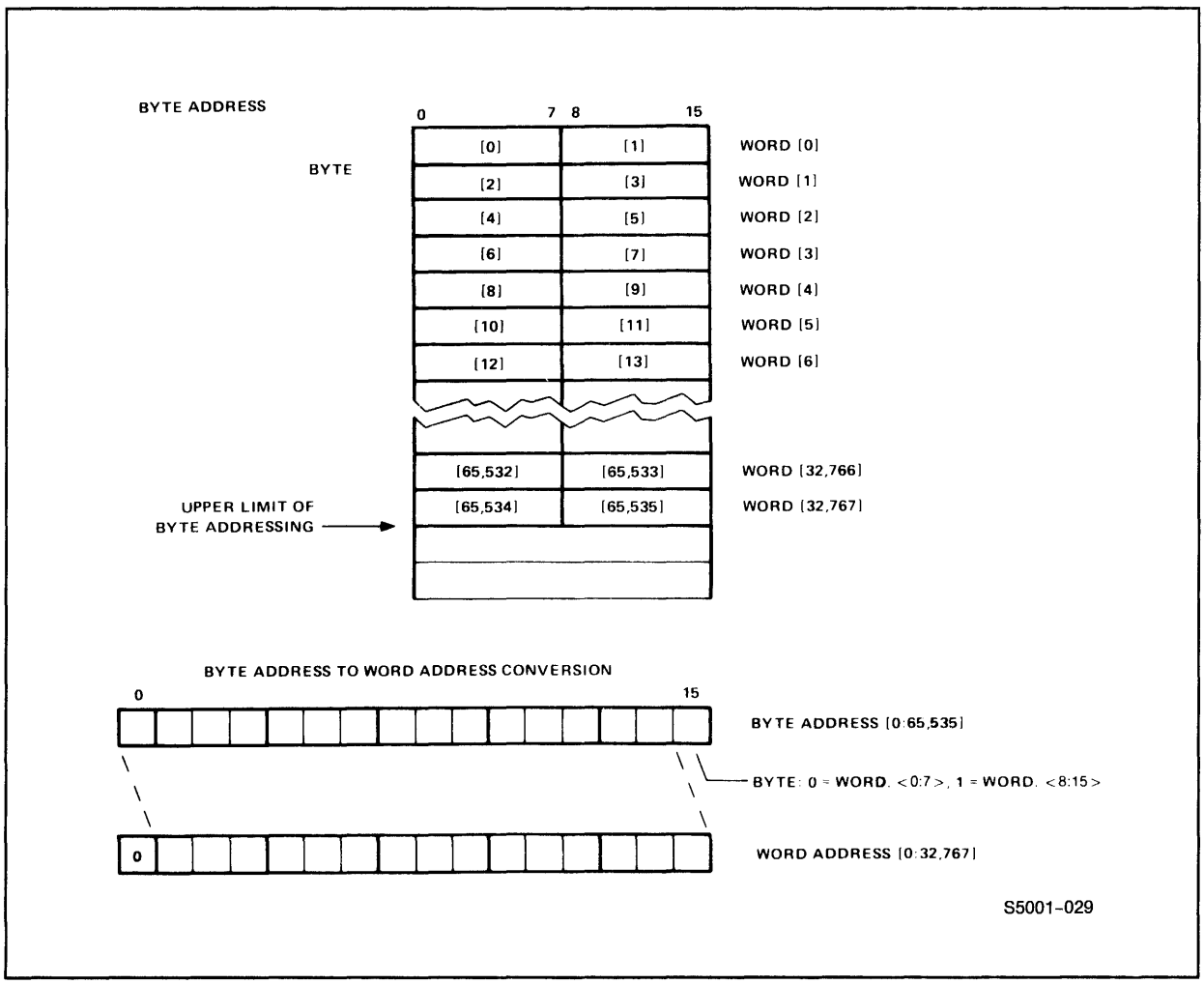


Figure 3-3. Byte Addressing

the byte address is extracted and used to specify left byte (0) or right byte (1); the remaining 15 bits are logically shifted right by one bit to form the word address. In addressing a byte in the code segment, bit 0 of the word address is copied from bit 0 of the P Register.

The following instructions are provided for referencing bytes in logical memory:

LDB	Load Byte into Register Stack from data segment
STB	Store Byte from Register Stack into data segment
LBP	Load Byte into Register Stack from Program (code segment)

Four instructions operate on blocks of bytes:

MOVB	Move Bytes from one memory location to another
COMB	Compare Bytes in one memory location with another
SBW	Scan a block of Bytes While a test character is encountered
SBU	Scan a block of Bytes Until a test character is encountered

Doublewords

Two 16-bit words can be accessed as a single 32-bit element. The hardware provides element access to doublewords in the data segment (the software simulates doubleword access to elements in the code segment). Doubleword elements are addressed on word boundaries; therefore doubleword addressing is permitted in all of the data segment.

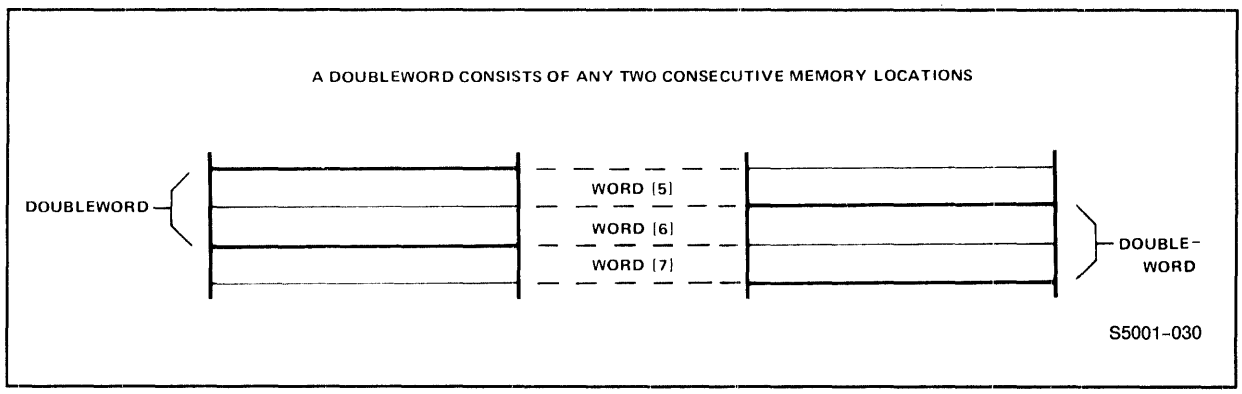


Figure 3-4. Doubleword Addressing

Two instructions are provided for referencing doublewords in logical memory:

LDD	Load Doubleword into Register Stack from data segment
STD	Store Doubleword from Register Stack into data segment

Quadruplewords

Four 16-bit words can be accessed as a single 64-bit element. The hardware provides element access to quadruplewords in the data segment (the software simulates quadrupleword access of elements in the code segment). Quadrupleword elements are addressed on word boundaries; therefore quadrupleword addressing is permitted in all of the data segment.

Two instructions are provided for referencing quadruplewords in the data segment:

QLD	Quadrupleword Load into Register Stack from data segment
QST	Quadrupleword Store from Register Stack into data segment

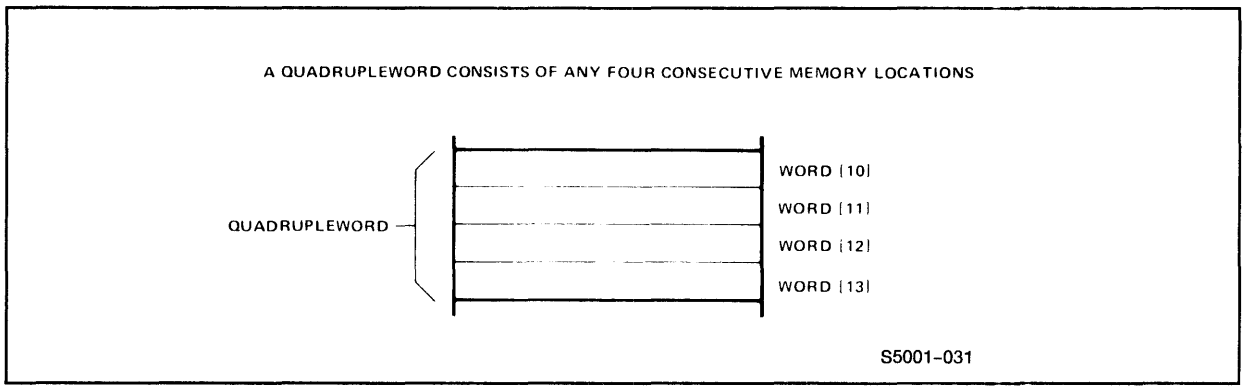


Figure 3-5. Quadrupleword Addressing

NUMBER REPRESENTATIONS

The system hardware provides arithmetic on both signed and unsigned numbers. Signed numbers are characterized by being able to represent both positive and negative values; unsigned numbers represent only positive values. Signed numbers are represented in 16 bits (a word), 32 bits (doubleword), or 64 bits (quadrupleword). Representation of unsigned numbers is restricted to 8- and 16-bit quantities.

Positive values are represented in true binary notation. Negative values are represented in two's-complement notation with the sign bit of the most significant word set to 1 (that is, WORD[0].<0>). The two's complement of a number is obtained by inverting each bit position in the number, then adding a 1. For example, in 16 bits, the number 2 is represented:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

and the number -2 is represented:

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

The representable range of numbers is determined by the sizes of operands (i.e., word, doubleword, and quadrupleword).

Single Word

Single-word operands can represent signed numbers in the range of

-32,768 to +32,767

and unsigned numbers in the range of

0 to +65,535

Whether a word operand is treated as a signed or an unsigned value is determined by the instruction used when a calculation is performed. Signed arithmetic is indicated by the execution of integer instructions. The integer instructions are:

IADD	Integer Add
ISUB	Integer Subtract
IMPY	Integer Multiply
IDIV	Integer Divide
INEG	Integer Negate (two's complement)
ICMP	Integer Compare
ADDI	(integer) Add Immediate

CMPI	(integer) Compare Immediate
ADM	(integer) Add to Memory

Unsigned arithmetic is indicated by the execution of logical instructions. The logical instructions are:

LADD	Logical Add
LSUB	Logical Subtract
LMPY	Logical Multiply (returns doubleword product)
LDIV	Logical Divide (returns 2-word quotient and remainder)
LNEG	Logical Negate (one's complement)
LCMP	Logical Compare
LADI	Logical Add Immediate

Doubleword

Doubleword operands can represent signed numbers in the range of

-2,147,483,648 to +2,147,483,647

Ten instructions perform integer arithmetic on doubleword operands. They are:

DADD	Doubleword Add
DSUB	Doubleword Subtract
DMPY	Doubleword Multiply
DDIV	Doubleword Divide
DNEG	Doubleword Negate (two's complement)
DCMP	Doubleword Compare
DTST	Doubleword Test
MOND	(load) Minus One in Doubleword form
ZERD	(load) Zero in Doubleword form
ONED	(load) One in Doubleword form

Byte

Byte operands represent unsigned values in the range of

0 to +255

This, of course, includes the ASCII character set. Byte operands are treated as the right half of word operands (that is, WORD.<8:15>) when arithmetic is performed (the left half of the word is assumed to be 0).

There is one instruction for testing the class (i.e., ASCII alphabetic, ASCII numeric, or ASCII special) of a byte operand. It is:

BTST	Byte Test

Quadrupleword (Decimal Arithmetic Option)

Quadrupleword operands for decimal arithmetic can represent 19-digit numbers in the range of

-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

NOTE

In this list, asterisks indicate optional instructions. Quadrupleword instructions not marked with an asterisk are part of the basic instruction set.

Six instructions perform integer arithmetic on quadrupleword operands:

QADD	Quadrupleword Add
QSUB	Quadrupleword Subtract
*QMPY	Quadrupleword Multiply
*QDIV	Quadrupleword Divide
*QNEG	Quadrupleword Negate
*QCMP	Quadrupleword Compare

Three instructions are provided for scaling (i.e, normalizing) and rounding quadrupleword operands:

QUP	Quadrupleword Scale Up
QDWN	Quadrupleword Scale Down
*QRND	Quadrupleword Round

Nine instructions are provided for converting operands between quadrupleword and other data formats:

*CQI	Convert Quadrupleword to Integer
*CQL	Convert Quadrupleword to Logical
*CQD	Convert Quadrupleword to Doubleword
*CQA	Convert Quadrupleword to ASCII
*CIQ	Convert Integer to Quadrupleword
*CLQ	Convert Logical to Quadrupleword
*CDQ	Convert Doubleword to Quadrupleword
*CAQ	Convert ASCII to Quadrupleword
*CAQV	Convert ASCII to Quadrupleword with Initial Value

Floating-Point and Extended Floating-Point

The fraction of a floating-point number is always normalized, to be greater than or equal to 1 and less than 2. The high-order integer bit is therefore dropped and assumed to have the value of 1. For all calculations, the sign is moved and the bit inserted. The integer plus 22 fraction bits of a floating-point number are equivalent to 6.9 decimal digits; the 55 bits of an extended floating-point number are equivalent to 16.5 decimal digits. If the value of the number to be represented is zero, the sign is 0, the fraction is 0, and the exponent is 0.

The fraction of a floating-point number is a binary number with the binary point always between the assumed integer bit and the high-order fraction bit. The exponent part of the number, bits 7 through 15 of the low-order word (see Figure 3-1), indicates the power of 2 multiplied by 1 plus the fraction. This field can contain values from 0 to 511. In order to express numbers of both large and small absolute magnitude, the exponent is expressed as an excess-256 value; that is, 256 is added to the actual exponent of the number before it is stored. The exponent range is therefore actually -256 through +255.

The sign of a floating-point number is explicitly stated in the high-order bit (i.e., signed magnitude representation). A 0 is positive, and a 1 is negative.

The absolute-value range of floating-point numbers is:

$$+/- 2^{-256} \quad \text{to} \quad +/- (1 - 2^{-23}) * 2^{256}$$

$$(\text{approx. } +/- 8.62 * 10^{-78}) \qquad (\text{approx. } +/- 1.16 * 10^{77})$$

For extended floating-point numbers, the range is the same; only the precision is increased:

$$+/- 2^{-256} \quad \text{to} \quad +/- (1 - 2^{-55}) * 2^{256}$$

$$(\text{approx. } +/- 8.62 * 10^{-78}) \qquad (\text{approx. } +/- 1.16 * 10^{77})$$

(Note, however, that the value +2**-256 is not representable; it would look like 0 in either floating point or extended floating point.)

Arithmetic

The result of integer arithmetic (IADD, ISUB, IMPY, DADD, DSUB, DMPY, QADD, QSUB) must be representable within the number of bits comprising the operand minus the sign bit (e.g., 15 bits for a word operand, 31 bits for a doubleword operand). If the result cannot be represented, an arithmetic overflow condition occurs, and no part of the results on the stack can be assumed valid. When an overflow occurs, the hardware Overflow indicator sets, and (if enabled) an interrupt to the operating system overflow interrupt handler occurs. An overflow condition also occurs if a divide operation is attempted with a divisor of 0.

The results obtained from a logical add or subtract (LADD or LSUB) are identical to that obtained from integer add or subtract, except that logical add and subtract do not set the Overflow indicator. The 16-bit result, the Condition Code setting, and the Carry indicator setting are the same. Logical divide (LDIV), however, sets the Overflow indicator if the quotient cannot be represented in 16 bits.

In addition to the Overflow indicator, two other hardware indicators are subject to change as the result of an arithmetic operation. They are:

- Condition Code (CC). This generally indicates if the result of a computation was a negative value, zero, or a positive value. (The Condition Code can be tested by one of the branch-on-condition-code instructions and program execution sequence altered accordingly.)

- Carry (K). This indicates that a carry out of the high-order bit position occurred.

For floating-point and extended floating-point arithmetic, the Overflow indicator is set if the exponent becomes either greater than +255 (exponent overflow) or less than -256 (exponent underflow) when trying to represent the normalized result of some operation. If the divisor in a divide operation is 0, the Overflow indicator is also set. If any conversion instruction causes a numeric overflow ("illegal conversion"), the Overflow indicator is set, and the result (including Condition Code) is undefined. If the result of some operation has a zero fraction and nonzero exponent or sign, the value is forced to zero.

Table 3-1 defines termination conditions for various floating-point arithmetic errors. (For further explanation of the Condition Code, refer to the "Environment Register" in the next section.)

Table 3-1. Floating-Point Error Terminations

Condition	Overflow	CC	Result
Exponent Overflow	1	00	Calculated result with error truncated
Exponent Underflow	1	10	
Divide by Zero	1	01	
Illegal Conversion	1	xx	Undefined

SECTION 4

INSTRUCTION PROCESSING ENVIRONMENT

A program executing as a process in a processor module consists of instruction codes in a code space in memory that manipulate variable data in a separate data segment in memory. The IPU's eight-element Register Stack is used to perform arithmetic operations and memory indexing. The instruction-to-instruction environment of a program is maintained in the IPU's Environment register. Programs themselves are separated into functional blocks of instructions called procedures.

These fundamental elements of the instruction processing environment are illustrated in Figure 4-1 and are discussed under separate subheadings below.

CODE SPACE

The code space of a given process consists of a user code space (UC) and optional library space (UL). Each space is a single program file that can contain up to 16 code segments. A space ID (space identifier) index is an octal number in range of 0 to %17 that is used to name the idividual segments within these two spaces (for example, UC.0, UC.5, UL.17). The IPU microcode keeps track of which segment is "current" for each space, and performs segment "switching" when necessary. External procedure calls are used to call procedures in other segments of the user space, as well as to call procedures in the system library.

Information in a code segment consists of instruction codes and program constants. Although it is possible to address the code segments (using extended addressing or the LBP, LWP, or LWUC instruction), only read access is permitted; a write access attempt results in an address trap. Therefore, the code segments cannot be modified during execution.

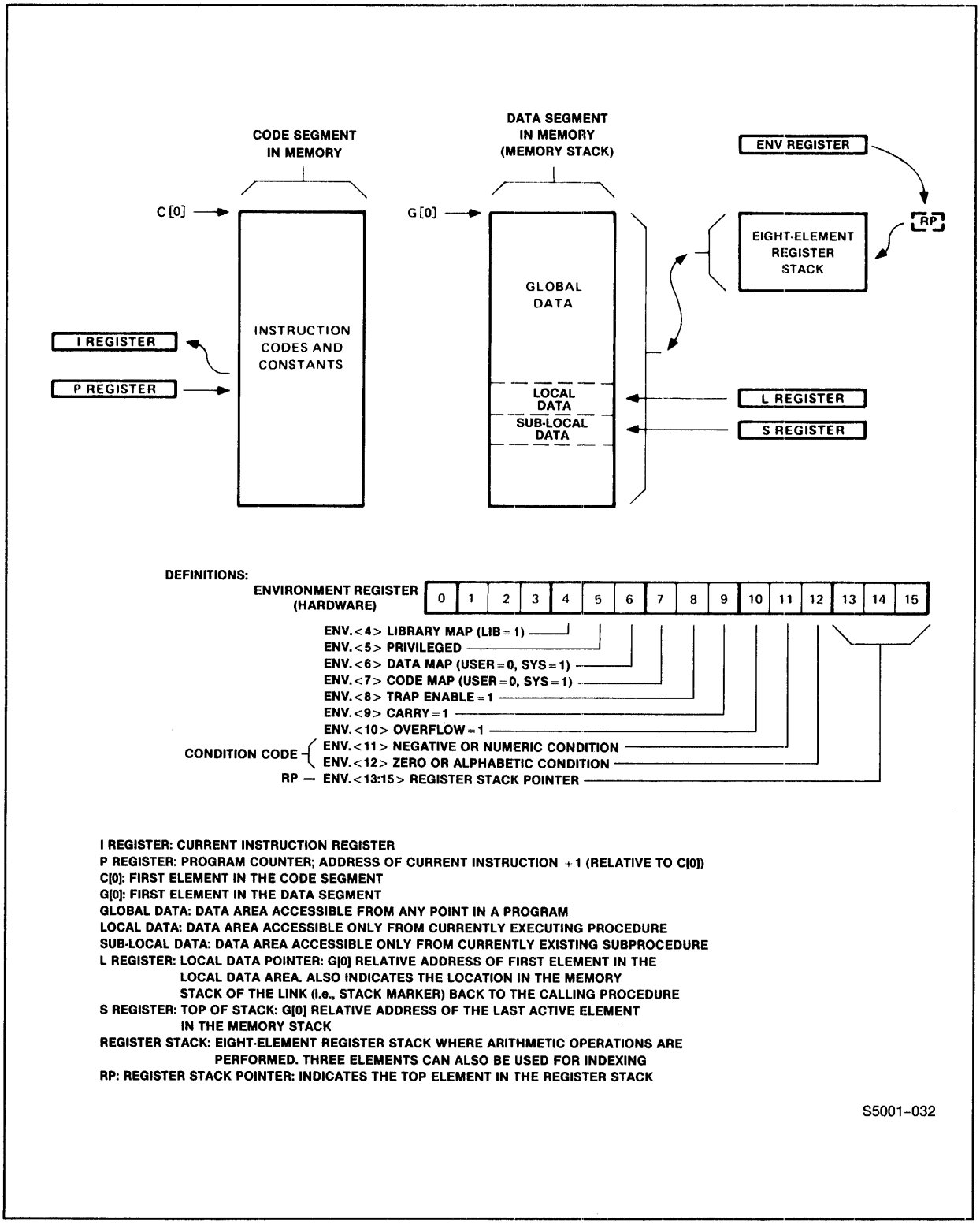


Figure 4-1. Elements of the Instruction Processing Environment

A code segment consists of up to 65,536 16-bit words. Words in a code segment are numbered consecutively from C[0] (code, element 0) through C[65,535]. This forms the basis for logical addressing within the code segment and is illustrated in Figure 4-2.

Two registers are associated with code segments. These are described in the following two paragraphs.

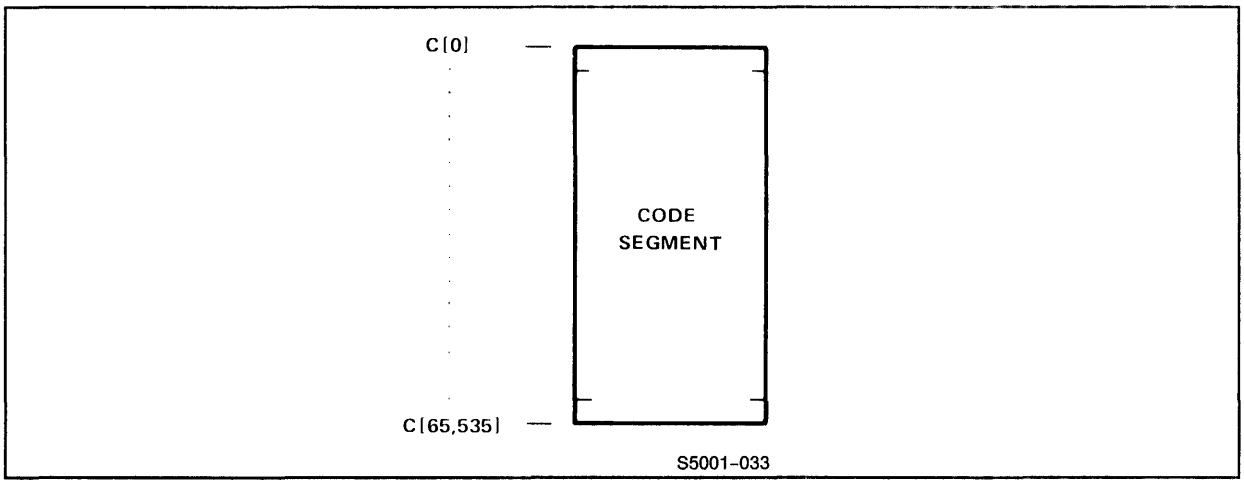


Figure 4-2. Code Segment Addressing Range

P register. The P (program) register is the program counter. It contains the 16-bit C[0]-relative address of the current instruction plus one. The contents of the P register are incremented by one at the beginning of instruction execution so that, nominally, instructions are fetched (and executed) from ascending memory locations. (See top diagram of Figure 4-3.)

When a program branch is taken, a procedure or subprocedure is called, or an interrupt occurs, the C[0]-relative address of the next instruction to be executed is placed in the P register. (See bottom diagram of Figure 4-3.)

I register. The I (instruction) register contains the machine instruction currently being executed. When the current instruction is completed, this 16-bit register is filled with the instruction in a code segment pointed to by the current setting of the P register. The contents of the P register are then incremented by one, as described above.

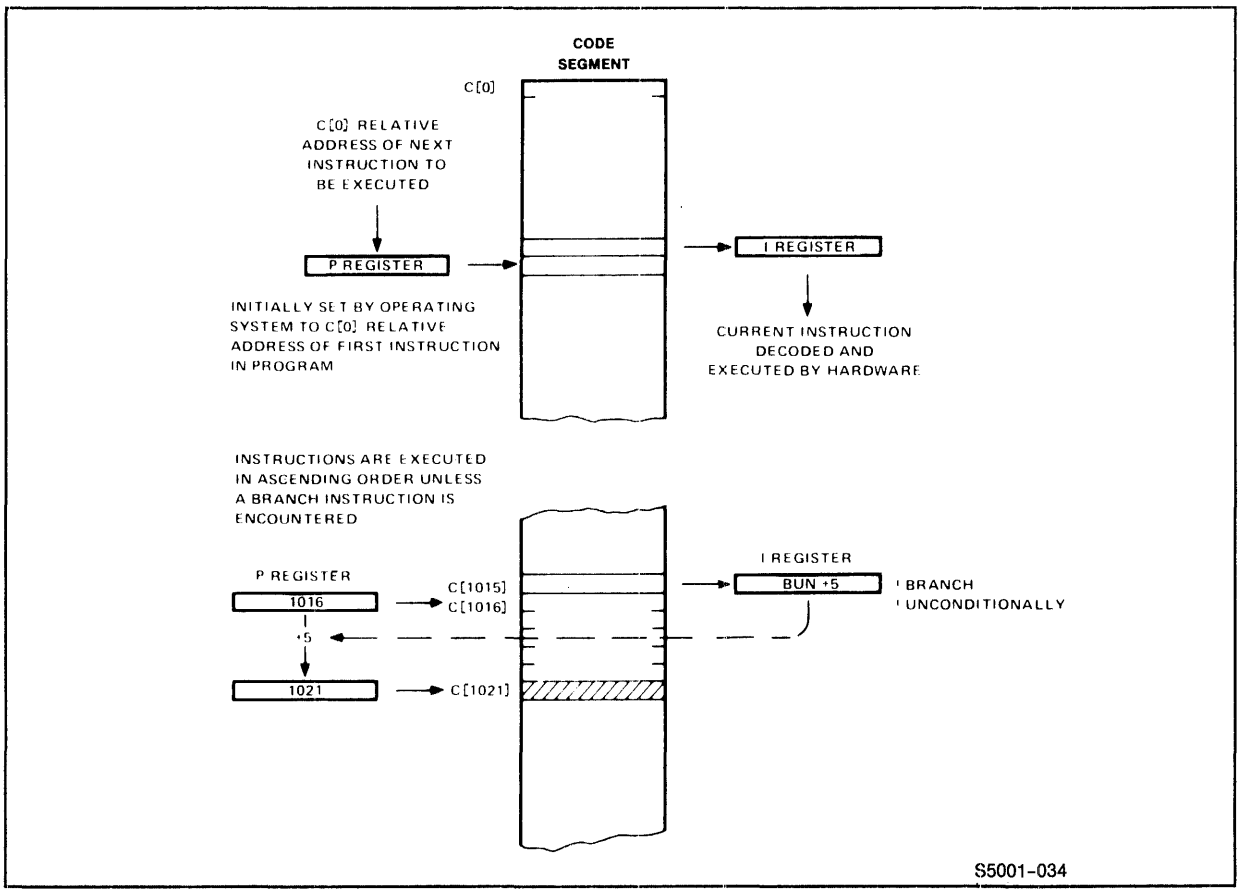


Figure 4-3. P Register and I Register

Addressing Code

Addresses for branching (and for constants) in a code segment are calculated relative to the current setting of the P register. This is referred to as self-relative addressing.

Instructions that reference a code segment have an eight-bit field for specifying a relative displacement from the current P register setting. The range of the displacement is therefore -128:+127 words. An example, the BUN instruction, is shown in Figure 4-4.

The location that is addressed by the displacement is referred to as the directly addressable location. It might be the location ultimately referenced by the instruction (that is, it might be the branch location, or it might contain the constant) or might itself contain a self-relative address. If the latter, then the

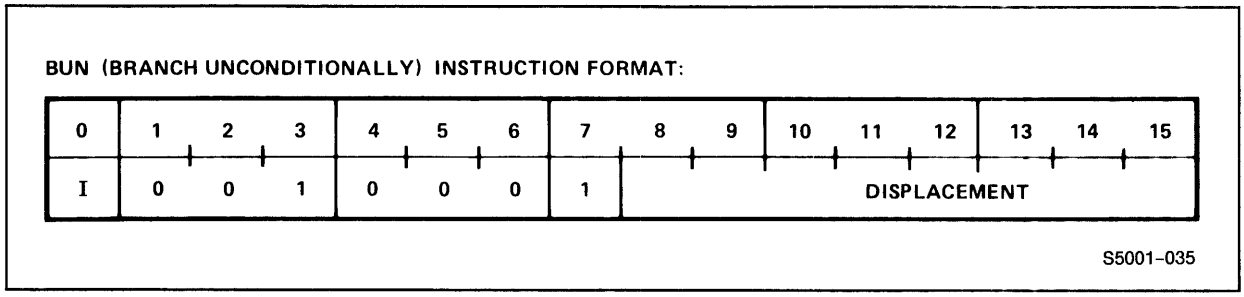


Figure 4-4. Displacement Field for Code Segment Instructions

referenced location is a relative displacement from the directly addressable location. This choice, whether the direct location is the one referenced by the instruction or contains a self-relative address, is specified by the indirect bit, <i>, in the instruction.

The address of the location in a code segment ultimately referenced by an instruction is called "branch^addrs" (branch address). This is the address placed in the P register when a program branch is taken:

```
P := branch^addrs;
   .
   .
I := code [ P ];       ! "code" refers to a code segment
```

and used when fetching a program constant from memory:

```
A := code [branch^addrs];
```

(A is the top element of the Register Stack.)

The address calculated by adding the displacement to the current P register setting is referred to as "dir^branch^addrs" (direct branch address):

```
dir^branch^addrs = P + <displacement>;
```

If the referenced location is within the range of the displacement (i.e., P [-128:+127]), then direct addressing is indicated, and the direct branch address is used as the branch address. If the referenced location is beyond the range of the displacement, then indirection is indicated, and the referenced location (branch^addrs) is a relative displacement from the direct branch address.

Direct addressing is specified when the <i> (indirection) bit, I.<0>, of the instruction is equal to 0; bits I.<8:15> are a two's-complement number (bit I.<8> is the sign bit) giving a positive or negative displacement from the current P register setting; therefore:

```
branch^addrs = dir^branch^addrs;
```

Indirect addressing is specified when the <i> bit of the instruction is equal to 1; bits I.<8:15> are a positive or negative displacement from the current P register setting; therefore:

```
branch^addrs = dir^branch^addrs + code [dir^branch^addrs];
```

That is, the C[0]-relative direct branch address is first calculated (a displacement from the current P register setting). Then the contents of the direct location (containing a displacement from itself) are added to the direct branch address. The result is the C[0]-relative branch address.

Examples of both direct and indirect addressing are given in Figure 4-5. The "I" in the LWP 9,I instruction signifies indirect addressing.

In addition to direct and indirect addressing, an offset value in a hardware register can be added to the address of the direct or indirect location before the final address is calculated. This permits a code segment location to be referenced as an offset from a base location; this is called indexing. Indexing in a code segment is discussed in Section 9, "Instruction Set," under the LWP instruction.

Addressing of byte elements (with indexing) is also permitted in the code segment, though restricted to only half of the segment (the same half in which the current P register setting is located). Byte addressing is discussed in Section 9 under the LBP (load byte from program) instruction.

By whatever means the final address is calculated, that address is the effective memory address.

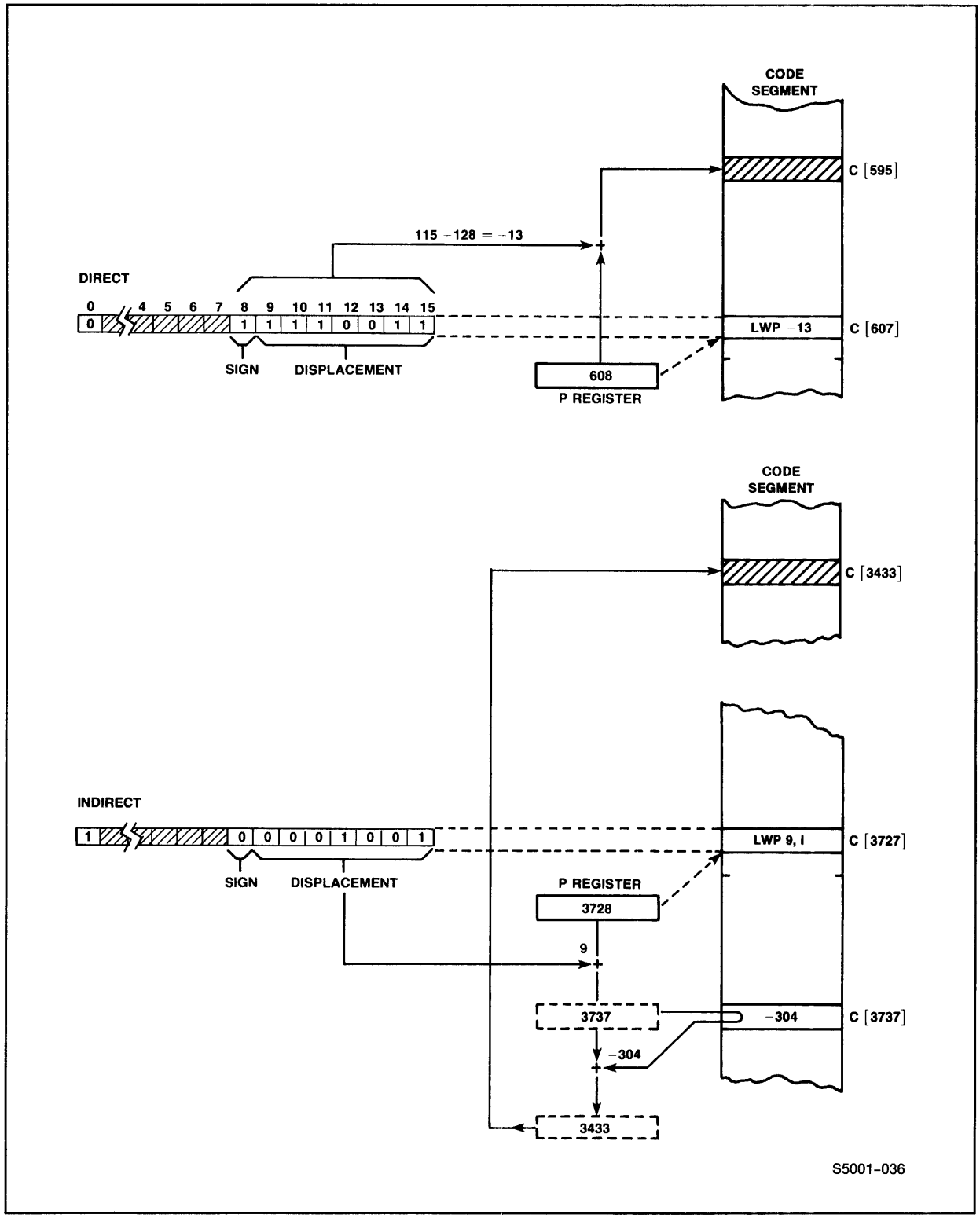


Figure 4-5. Addressing in a Code Segment

DATA SEGMENT

Data Storage and Access

The data segment contains a program's temporary storage locations (i.e., variables). Information in this segment consists of single-element items, multiple-element items (arrays), and address pointers. Input/output transfers (which are performed on behalf of application programs by the GUARDIAN file system) are accomplished using arrays in a program's data segment.

The first half of the data segment is used for dynamic allocation of storage when procedures are invoked (see "Procedures"); this area is referred to as the memory stack.

The data segment consists of up to 65,536 16-bit words. Addresses in the data segment start at G[0] (global data, word 0) and progress consecutively through G[65,535]. See Figure 4-6. The memory stack portion of the data segment is G[0:32,767].

Data is accessed through use of the memory reference instructions. Locations in the data segment are addressed either through the address field in a memory reference instruction (direct addressing) or through an address pointer in memory (indirect addressing). Additionally, the memory reference instructions permit an offset value (in a hardware register) to be added to a direct or indirect address before a final address is calculated. This permits one data element to be referenced as an offset from another data element (indexing).

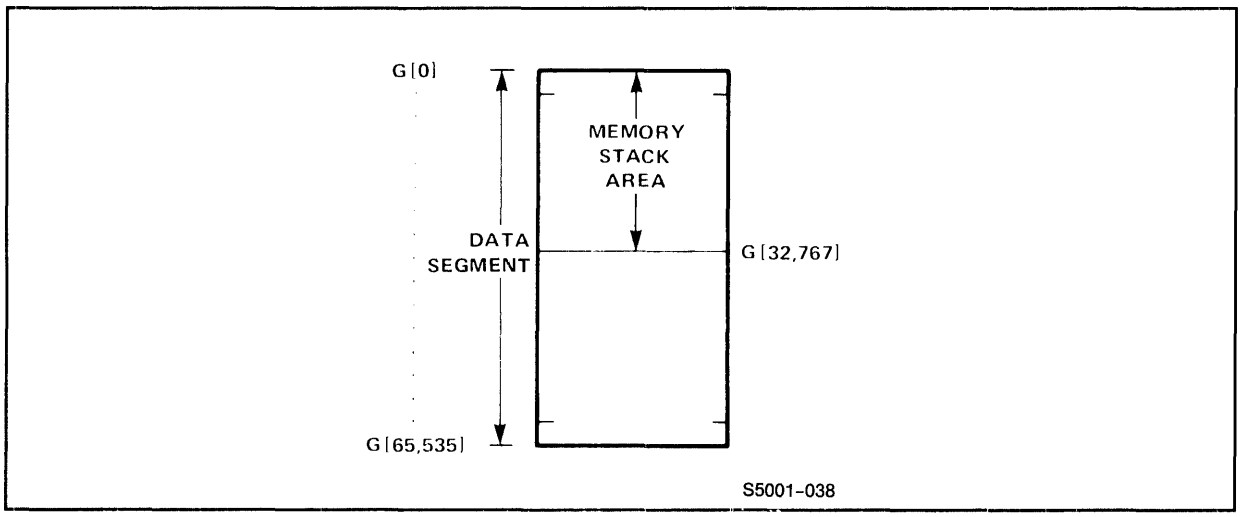


Figure 4-6. Data Segment Addressing Range

Direct addressing permits only limited ranges of addressing within the data stack area; these ranges are defined under the next subheading ("Addressing Data"). Indirect addressing and indexing permit access to the entire data segment, since an entire 16-bit word is used to specify an address. Memory reference instructions have only 5 to 8 bits (depending on the instruction) to specify a direct address.

The memory reference instructions for the data segment are:

LDX	Load Index register from data segment
NSTO	Nondestructive Store, Register Stack into data segment
LOAD	Load word into Register Stack from data segment
STOR	Store word from Register Stack into data segment
LDB	Load Byte into Register Stack from data segment
STB	Store Byte from Register Stack into data segment
LDD	Load Doubleword into Register Stack from data segment
STD	Store Doubleword from Register Stack into data segment
ADM	Add to Memory

The memory stack portion of the data segment is logically separated into three areas: global, local, and sublocal (or "top-of-stack" area). Each logical area has an addressing base so that relative addressing can be performed. The logical areas are described in the following paragraphs and illustrated in Figure 4-7.

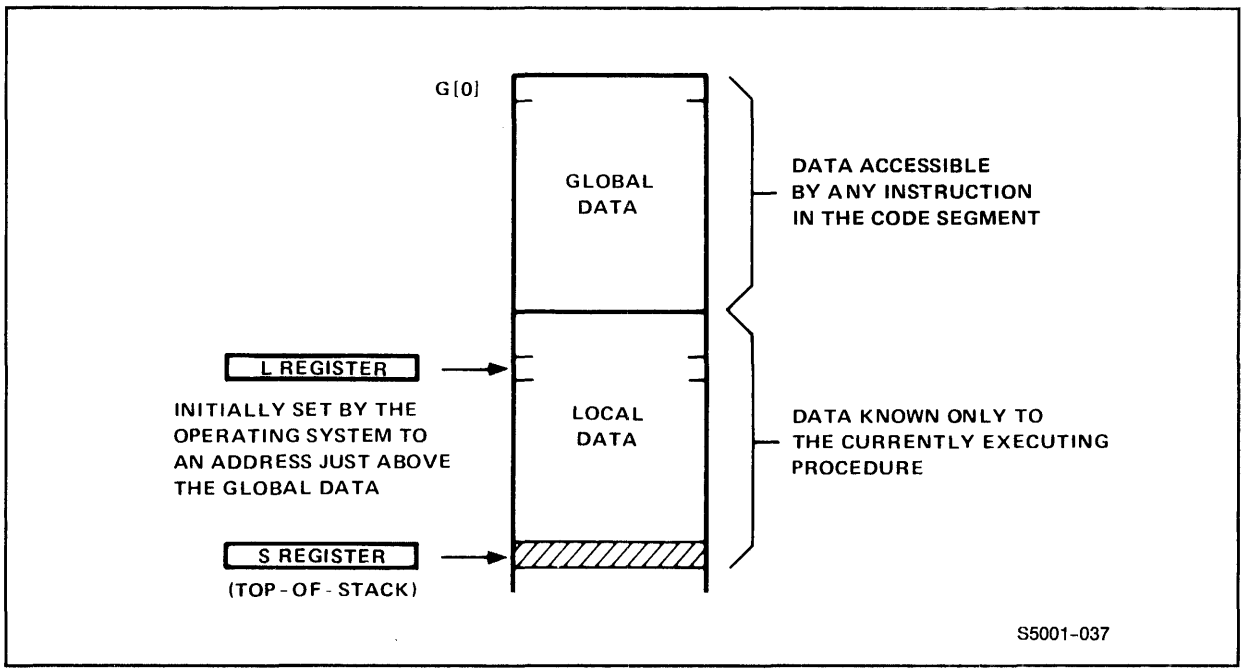


Figure 4-7. L Register and S Register

Global Area. Data within the global area is addressable by any instruction in the program. The addressing base of the global area is defined as G[0].

The beginning of the global area coincides with the beginning of the data segment. Thus, the G[0]-relative address of an item is its logical address within the data segment. G[0] is logical address 0.

Local Area. Data within the local area is known only to the currently executing procedure. The local area is defined by the 16-bit L register. The L (for local) register contains the G[0]-relative address of the word at the beginning of this area. The addressing base of the local area is defined as L[0].

When a procedure is called, a new local area is defined. This occurs because the address contained in the L register advances to point above the current local area (the caller's local area is then undefined). Conversely, when a procedure exits, the exiting procedure's local area is deleted (and the preceding local area redefined) because the address in the L register recedes back to its previous setting.

Top-of-Stack Area. Data in the top-of-stack (or sublocal) area is known only to the currently executing procedure. The top-of-stack location is defined by the 16-bit S register, which contains the G[0]-relative address of the last word currently defined in the memory stack. The addressing "base" of the top-of-stack area is defined as S[0], and the sublocal area consists of up to 32 word locations including and preceding S[0].

During execution of a procedure, the address in the S register advances as elements are moved from the Register Stack to the top of the memory stack, and recedes as elements are moved from the top of the memory stack to the Register Stack. The address also advances when procedures and subprocedures are invoked and recedes when they are exited, along with the L register address.

Addressing Data

Data elements in the data segment are fetched and stored by the hardware in terms of word addresses, regardless of the type of operand involved. (The instruction set microcode also provides for the addressing of bytes within a word, as described in the sections on "Direct Addressing" and "Indirect Addressing" that

follow.) For purposes of explanation, "data" refers to a data segment and "address" refers to the G[0]-relative address of a word referenced by an instruction. Together, "data" and "address" are used to indicate access to a location in a data segment referenced by an instruction. Thus, a LOAD instruction into A (the top of the Register Stack) is:

```
A := data [address];
```

All direct addressing in the data segment is relative to one of the three addressing bases: G[0], L[0], or S[0]. Memory reference instructions for data contain a 9-bit address field for specifying one of the three addressing bases and a relative displacement from that base. Four addressing modes are provided for addressing relative to these bases. The address indicated by the address field in a memory reference instruction is referred to as the direct^address. The addressing modes are: G-relative, L-plus-relative, L-minus-relative, and S-minus-relative. These are described in the following paragraphs. Figure 4-8 shows an example of a memory reference instruction and defines the bit patterns for the four addressing modes. Figure 4-9 illustrates each of the addressing modes.

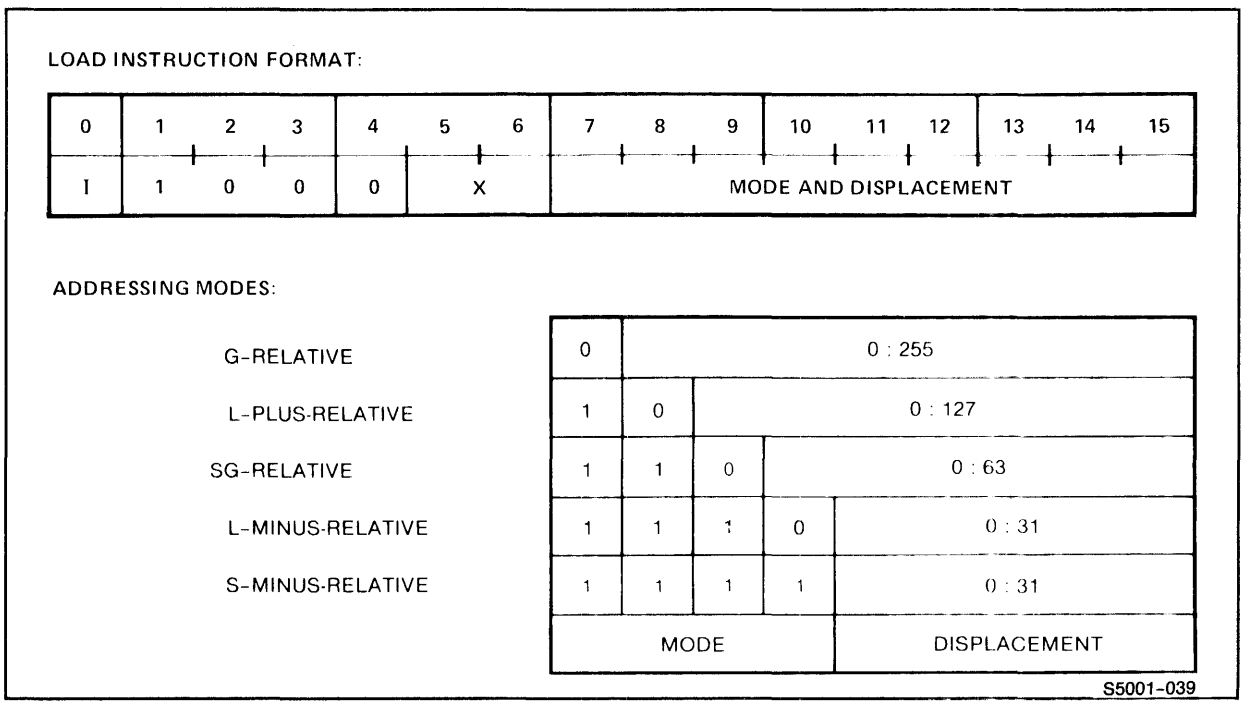


Figure 4-8. Mode and Displacement Field for Memory Reference Instructions

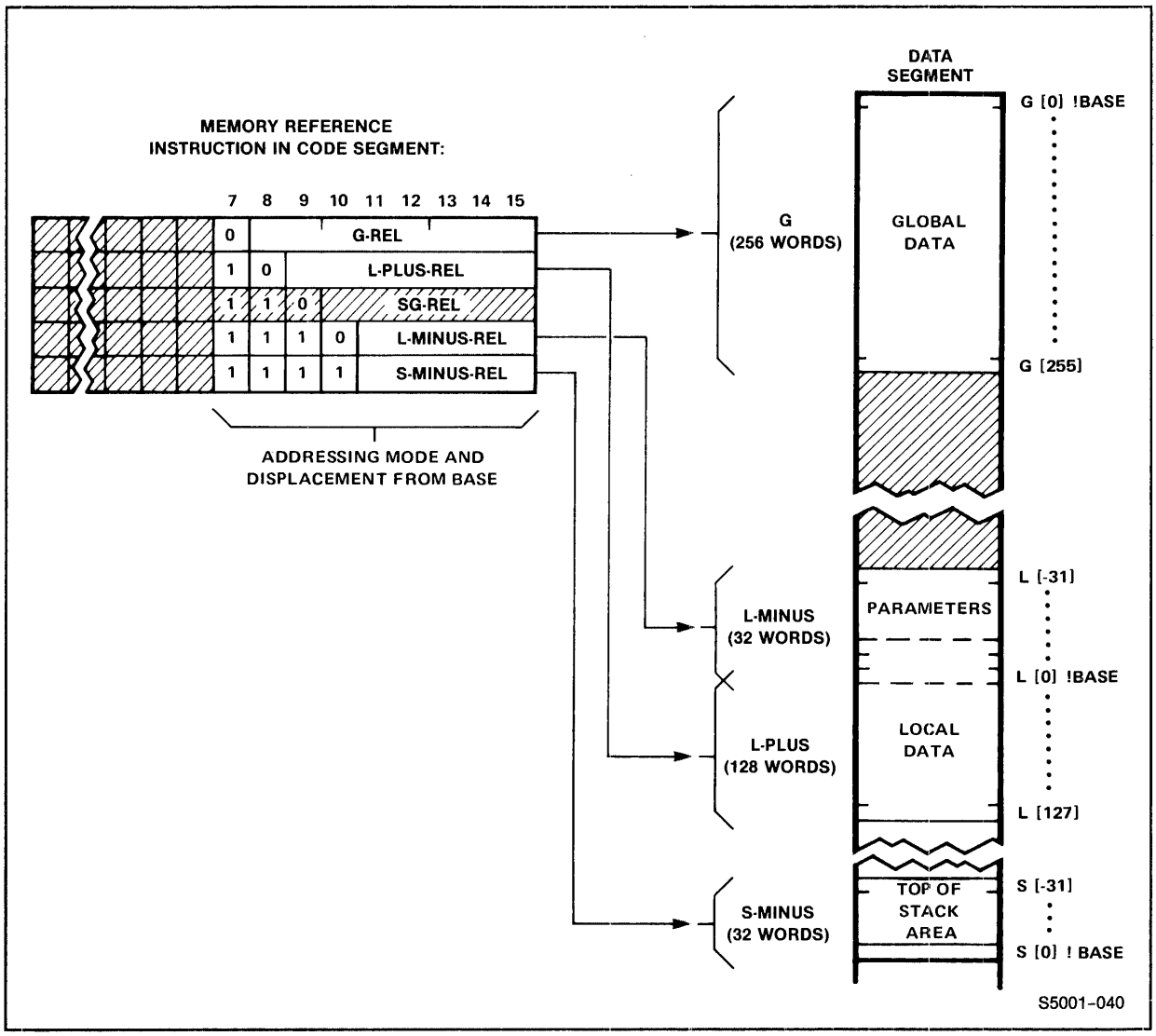


Figure 4-9. Memory Reference Instruction Addressing Modes

- G-Relative Mode

  This mode addresses the first 256 locations in the global area (G[0:255]). The G-relative mode is indicated when bit I.<7> of a memory reference instruction is equal to 0; bits I.<8:15> specify a positive word displacement from G[0]; that is:

  ```
  direct^address := I.<8:15>;
  ```

- L-Plus-Relative Mode

  This mode addresses the first 128 words of a procedure's local data area (L[0:127]). The L-plus-relative mode is indicated when bits I.<7:8> of a memory reference instruction are equal to 10 (binary); bits I.<9:15> specify a positive word displacement from the current L[0].

  The hardware calculates a G[0]-relative address by adding I.<9:15> to the contents of the L register:

  ```
  direct^address := L + I.<9:15>;
  ```

- L-Minus-Relative Mode

  This mode addresses the 32 words just below and including the word pointed to by the current L register setting, L[-31:0]. (This area is used for procedure parameter passing.) The L-minus-relative addressing mode is indicated when bits I.<7:10> of a memory reference instruction are equal to 1110 (binary); bits I.<11:15> are a negative word displacement from the current L[0]. The hardware calculates a G[0]-relative address by subtracting I.<11:15> from the contents of the L register:

  ```
  direct^address := L - I.<11:15>;
  ```

- S-Minus-Relative Mode

  This mode addresses the 32 words just below, and including, the current top-of-stack word (S[-31:0]). (This area is used for a subprocedure's sublocal data and for temporary storage of the Register Stack contents by the PUSH and POP instructions). The S-minus-relative mode is indicated when bits I.<7:10> of a memory reference instruction are equal to 1111 (binary); bits I.<11:15> are a negative word displacement from the current S[0]. The hardware calculates a G[0]-relative address by subtracting I.<11:15> from the contents of the S register:

  ```
  direct^address := S - I.<11:15>;
  ```

An additional addressing mode is provided to access the system data segment from the user environment--the SG-Relative mode (see "Environment Register" for an explanation of user environment). This mode addresses the first 64 locations of the system data segment (SG[0:63]) and is usable only by procedures executing in privileged mode (e.g., the operating system). The SG-relative addressing mode is indicated when bits I.<7:9> of a memory

reference instruction are equal to 110 (binary). Bits I.<10:15> are a positive word displacement from SG[0]. (See "Calling External Procedures" later in this section for an explanation of SG-relative addressing.)

Direct Addressing. If the <i> (indirection) bit, I.<0>, of a memory reference instruction is equal to 0, then direct addressing is specified. The ranges of directly addressable locations in the data segment are:

```
G[0:255]   256 words  G-Relative Mode
L[0:127]   128 words  L-Plus-Relative Mode
L[-31:0]    32 words  L-Minus-Relative Mode
S[-31:0]    32 words  S-Minus-Relative Mode
```

With direct addressing, the address of an operand referenced by an instruction, relative to one of the addressing bases, is specified in the address field of the memory reference instruction; therefore,

```
address := direct^address;
```

and only one memory reference is needed to access the referenced memory location. Figure 4-10 gives an example of direct addressing.

If a byte operand is referenced, it is in the left half of the referenced location:

```
byte := data [address].<0:7>;
```

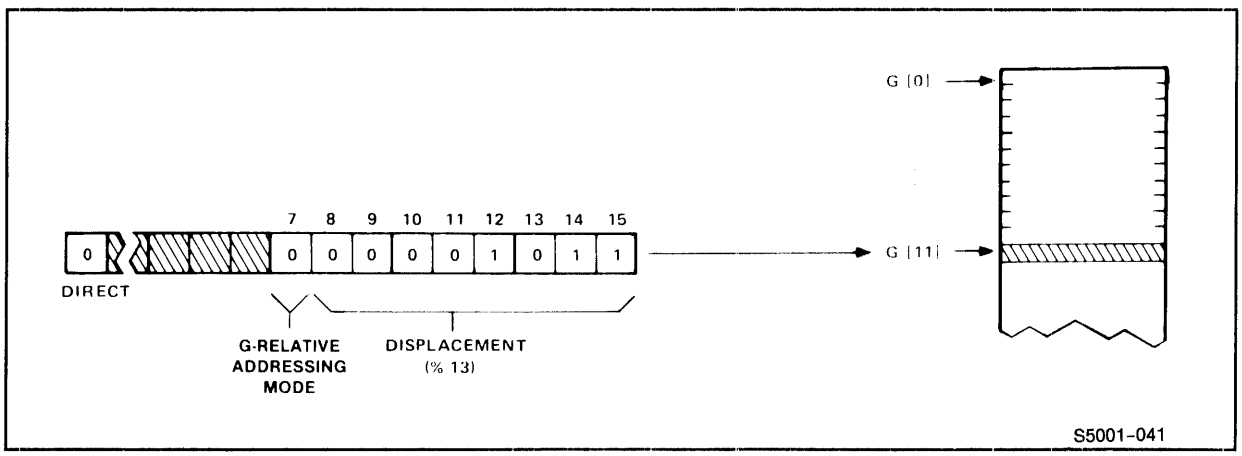


Figure 4-10. Direct Addressing in the Data Segment

If a doubleword operand is referenced, it consists of two words starting at the referenced location:

```
doubleword := data [address:address+1];   ! two words
```

Quadruplewords cannot be accessed as such by any of these modes. A quadrupleword must be accessed as some combination of smaller units, such as two doublewords or four words.

Indirect Addressing. If the <i> (indirection) bit, I.<0>, of a memory reference instruction is equal to 1, then indirect addressing is specified. The range of indirect addressing is G[0:65,535] (i.e., any location in the data segment).

With indirect addressing, the address of the referenced location, relative to G[0], is contained in a location that can be addressed directly (the contents of the direct location are referred to as an address pointer). Two memory references are needed to access the referenced location; the first to fetch the address,

```
address := data [direct^address];
```

the second to access the operand. Figure 4-11 gives an example.

If a byte operand is accessed, the address pointer contains a G[0]-relative byte address. Bits <0:14> of the address pointer are the word address of the byte operand, bit <15> of the address pointer indicates whether the referenced byte is in the left-hand part of the word, <0:7>, or the right-hand part, <8:15>:

```
byteaddress := data [direct^address];

address := byteaddress.<0:14>;
```

and the referenced byte is

```
byte := if byteaddress.<15> then
           data [address].<8:15>  ! right byte
        else
           data [address].<0:7>;  ! left byte
```

An example is shown in Figure 4-12.

Note that, because a byte address is effectively divided by two (to provide a word address), and the maximum byte address is 65,535, addressing of bytes is limited to the lower 32,768 words of a data segment (the memory stack area).

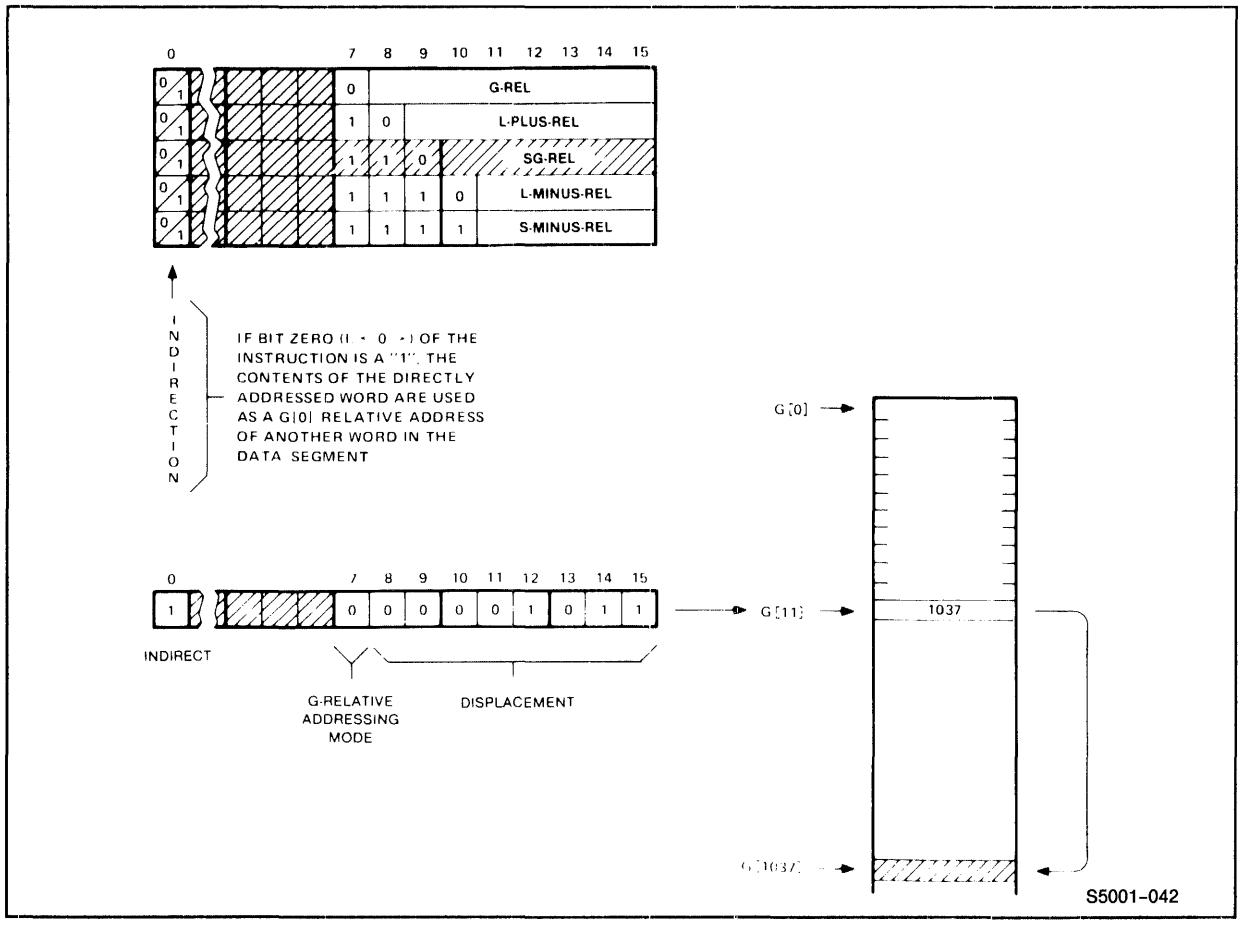


Figure 4-11. Indirect Addressing in the Data Segment

If a doubleword operand is accessed, the address pointer contains a G[0]-relative word address:

```
address := data [direct^address];
```

and the referenced doubleword is

```
doubleword := data [address:address+1];
```

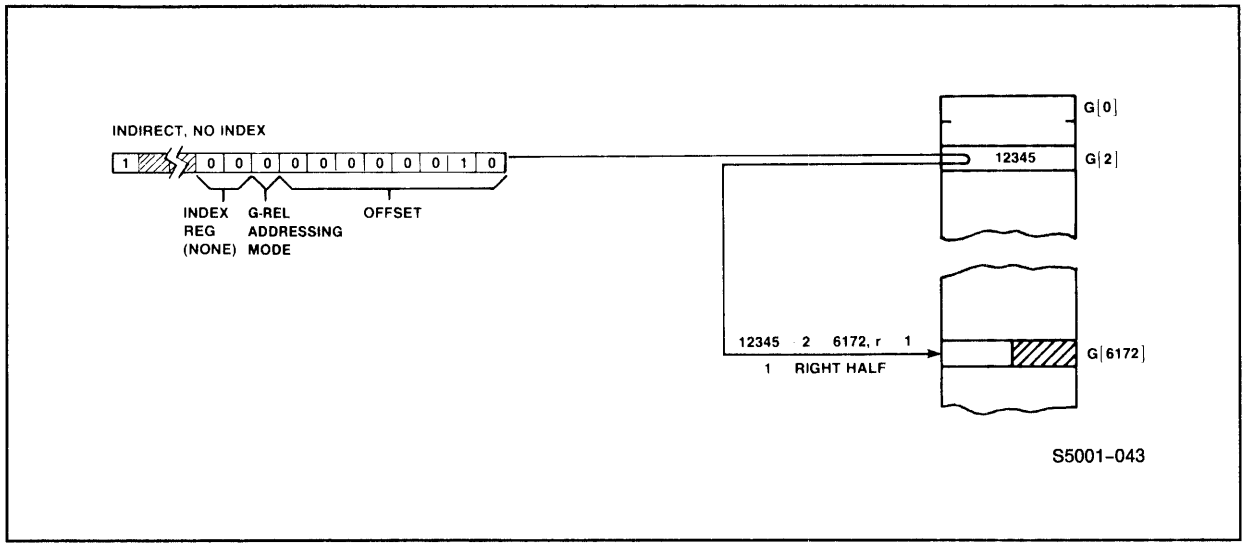


Figure 4-12. Indirect Byte Addressing in the Data Segment

Indexing. Indexing is used to reference memory locations relative to a data element in memory. A typical use is when an element in an array is accessed.

Generally, indexing is done as follows. An initial address is first calculated as described previously (any addressing mode as well as direct and indirect addressing is permitted). This initial address is then used as a base address for indexing. The indexing value, contained in an index register (referred to as "X"), is added to the initial address to provide the address of the referenced operand. This is shown in the upper part of Figure 4-13.

Any one of three registers in the Register Stack (R[5:7]) can be used as index registers. The register to be used for indexing is specified in the <x> (index) field, I.<5:6>, that is part of all memory reference instructions. (Note the instruction format in the lower part of Figure 4-13.) The index field corresponds to Register Stack elements as follows:

I.<5:6> VALUE	INDEX REGISTER
0	X = no indexing
1	X = R[5]
2	X = R[6]
3	X = R[7]

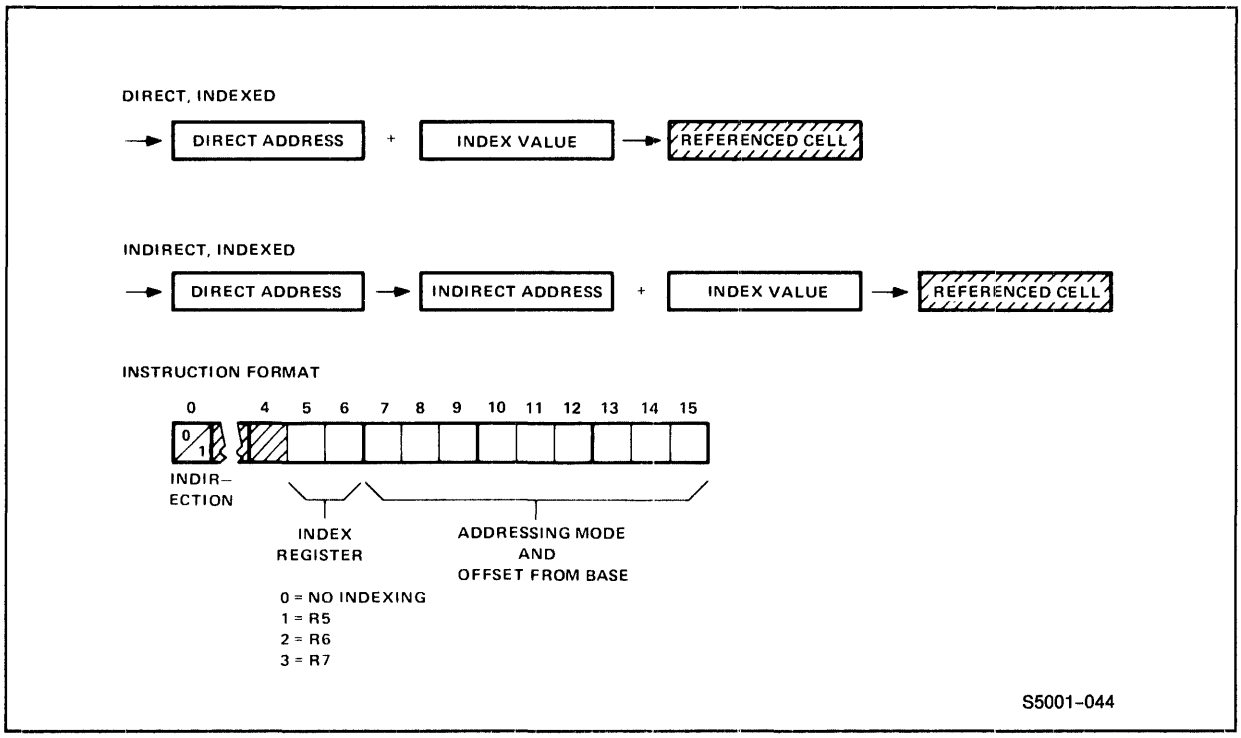


Figure 4-13. Indexing

An index register can contain values from -32,768 through +32,767 to provide direct word and doubleword addressing of any location in the data area (all addressing is modulo 65,535). The value in an index register is always treated as an element indexing value. That is, if a byte instruction is being executed, the contents of an index register are treated as a byte offset; if a doubleword instruction is being executed, the contents are treated as a doubleword offset.

Specifically,

- For direct, indexed addressing of word operands,

  ```
  address := direct^address + X;
  ```

  The contents of the index register, X, are added to the direct address; and the referenced element (referred to as "wordx") is:

  ```
  wordx := data [address];
  ```

- For indirect, indexed addressing of word operands,

```
address := data [direct^address] + X;

wordx := data [address];
```

- For direct, indexed addressing of byte operands,

```
byteaddress := (2 * direct^address) + X;
```

The direct^address (a word address) is multiplied by two to obtain a byte address. The indexing value (a byte offset) is added to that. The G[0]-relative address of the referenced byte is converted to a word address as follows:

```
address := byteaddress.<0:14>;
```

and the referenced byte (referred to as "bytex") is

```
bytex := if byteaddress.<15> then
            data [address].<8:15>  ! right byte
         else
            data [address].<0:7>;  ! left byte
```

- For indirect, indexed addressing of byte operands,

```
byteaddress := data [direct^address] + X;
```

The address pointer indicated by "data [direct^address]" contains a byte address. X, which contains a byte offset, is added to the byte address. The "address" and "bytex" are then determined as described above.

- For direct, indexed doubleword operands,

```
address := direct^address + (2 * X);
```

That is, the indexing value (a doubleword element index) is multiplied by two to provide a word index. This value is added to the initial address (also a word address) to generate a G[0]- relative word address, and the element referenced (referred to as "dwordx") is

```
dwordx := data [address:address+1];   ! two words
```

- For indirect, indexed doubleword operands,

```
address := data [direct^address] + (2 * X);
```

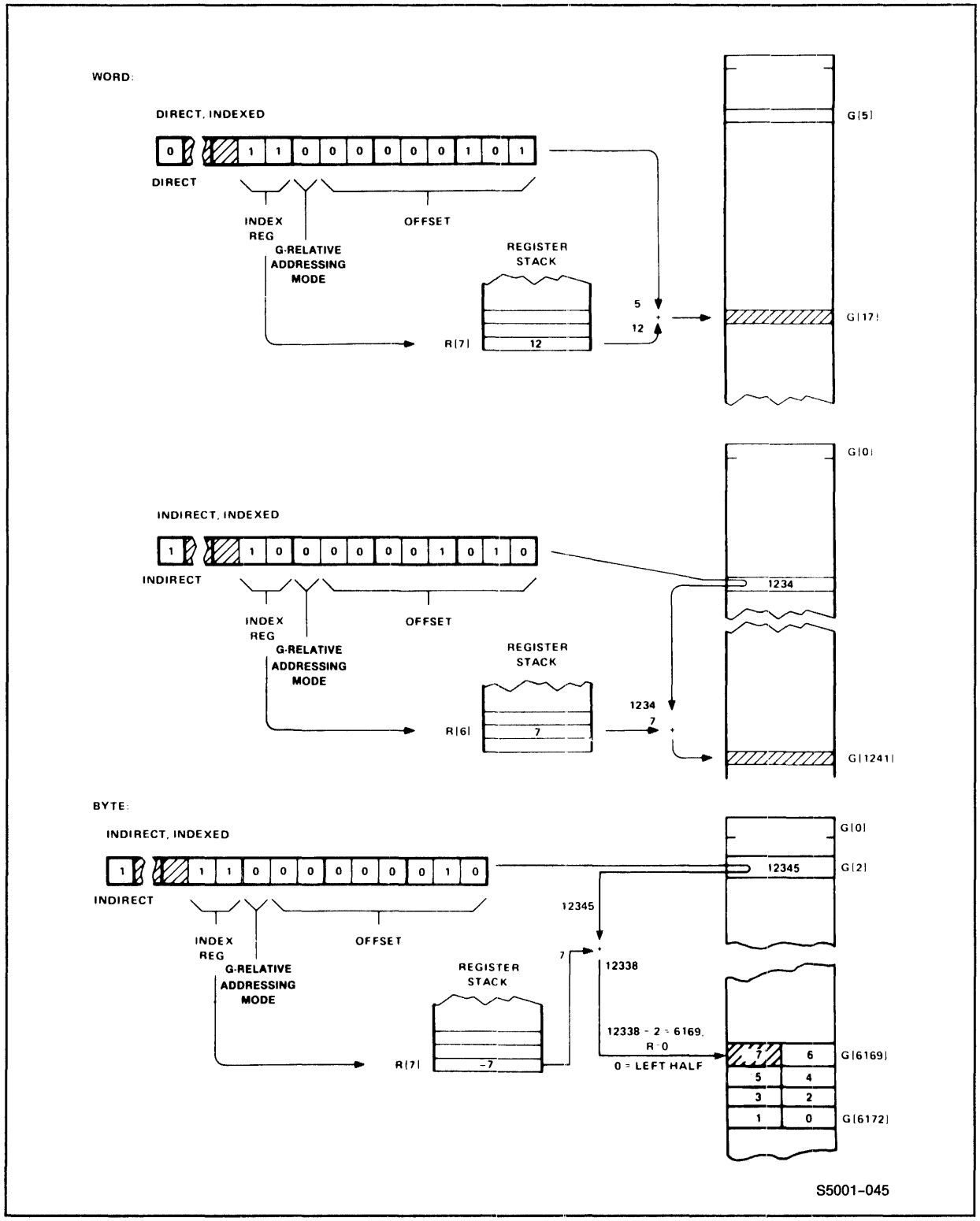


Figure 4-14. Examples of Indexing

The address pointer indicated by "data [direct^address]" contains a word address. X, which contains a doubleword offset, is multiplied by two (to generate a word offset) and added to the initial address. The "dwordx" is the same as described above.

Figure 4-14 shows examples of word and byte indexing.

Three instructions deal with loading and modifying index register contents. They are:

LDX	Load an Index register from data segment
LDXI	Load an Index register with Immediate operand
ADXI	Add to an Index register the Immediate operand

An additional instruction is used for branching on the contents of an index register. It is:

BOX	Branch on Index register less than A (top of Register Stack) and increment index register

REGISTERS

Register Stack

The Register Stack is where arithmetic computations are performed and, except for the Compare Words and Compare Bytes instructions, where comparisons are made. The Register Stack consists of eight 16-bit registers, designated R[0] (Register Stack, element 0) through R[7]; see Figure 4-15. Three elements of the Register Stack, R[5:7], also double as index registers (see "Indexing").

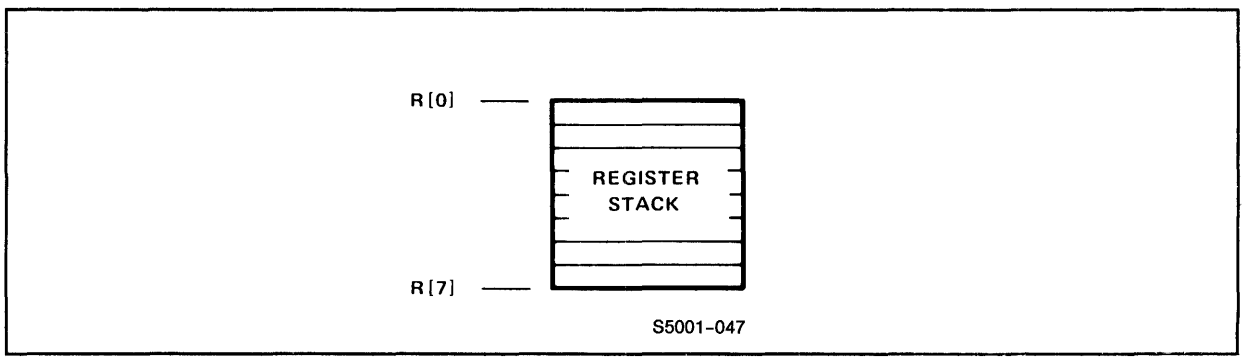


Figure 4-15. Register Stack

A typical operation to add two numbers in the Register Stack is as follows: the operands are first loaded into the Register Stack using LOAD instructions, an IADD (integer add) instruction is then executed performing the desired arithmetic, and the result is then stored back into memory using a STOR instruction. Grouped together to form a program, the preceding operation looks like this:

```
LOAD G + 002 ! load data element G[2] onto Register Stack
LOAD G + 003 ! load data element G[3] onto Register Stack
IADD         ! integer add
STOR G + 004 ! store result from Register Stack into G[4]
```

The condition of the Register Stack for each of these instructions is shown in Figure 4-16.

Usually, elements in the Register Stack are addressed implicitly. That is, an instruction operates on the top element (or elements) without specifying the actual registers involved. The current top element of the Register Stack is defined by the Register Stack Pointer, RP. RP, which is a three-bit field in the Environment register (described in the next subsection), contains the register number, 0:7, of the top element. The RP setting is incremented when operands are loaded into the Register Stack:

```
RP := RP + <size of element>;
```

and decremented when arithmetic is performed or results are stored:

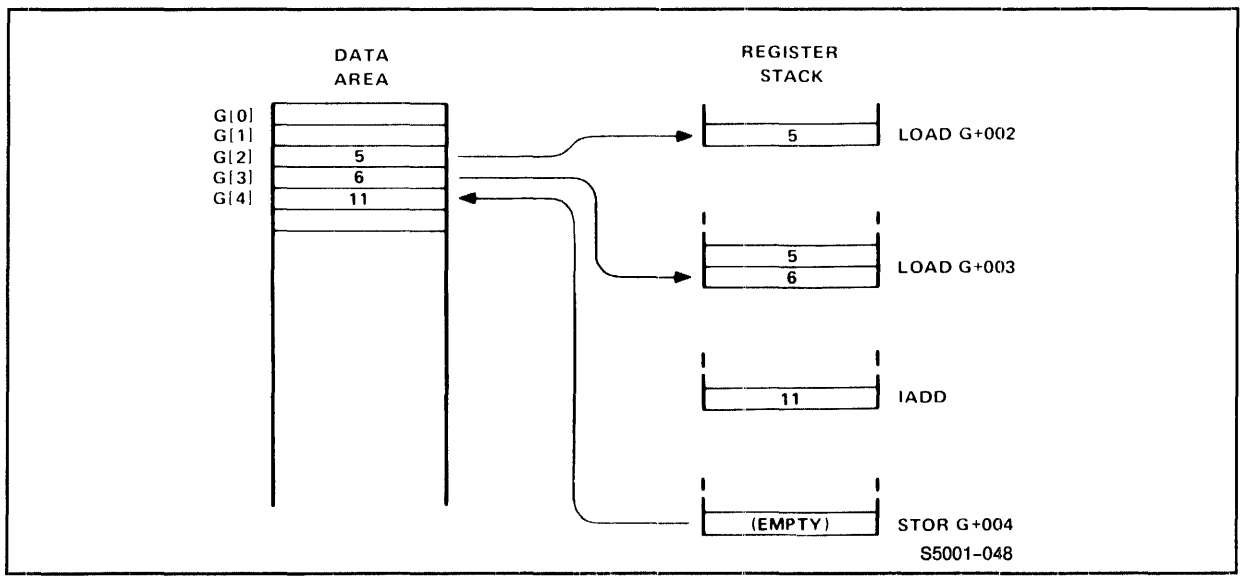


Figure 4-16. Example of Register Stack Operation

```
RP := RP - <size of element>;
```

The empty state of the Register Stack is defined as RP = 7. The full state is also RP = 7. There is no protection against rolling RP over from 7 to 0.

The operation of the Register Pointer for the example of Figure 4-16 is shown in Figure 4-17.

The elements in the Register Stack are named as to their location relative to the current top element. The top element is designated "A", the second is "B", and so on through "H":

```
A  = RP          (top of Register Stack)
B  = RP [-1]
C  = RP [-2]
D  = RP [-3]
E  = RP [-4]
F  = RP [-5]
G  = RP [-6]
H  = RP [-7]
```

Examples of register naming are shown in Figure 4-18.

Environment Register

The 16-bit ENV (Environment) register maintains the IPU state of the currently executing process. The individual bits and bit fields of the ENV register are continually referenced and updated by the IPU hardware and firmware. The ENV register contents are saved (along with the contents of the P and L registers) by the firmware as part of the executing state of a process when a procedure is invoked or when an interrupt occurs. The firmware restores the ENV register to its previous state when the procedure or interrupt finishes.

The format of the ENV register is shown in Figure 4-19. The following paragraphs describe the meanings of the bits in this register. (The four high-order bits are reserved for use as flags by the microcode.)

NOTE

The stored copy of the ENV register in a stack marker differs from the hardware format shown here, since the IPU microcode uses ENV.<11:15> to save the space ID index; compare with Figure 4-24.

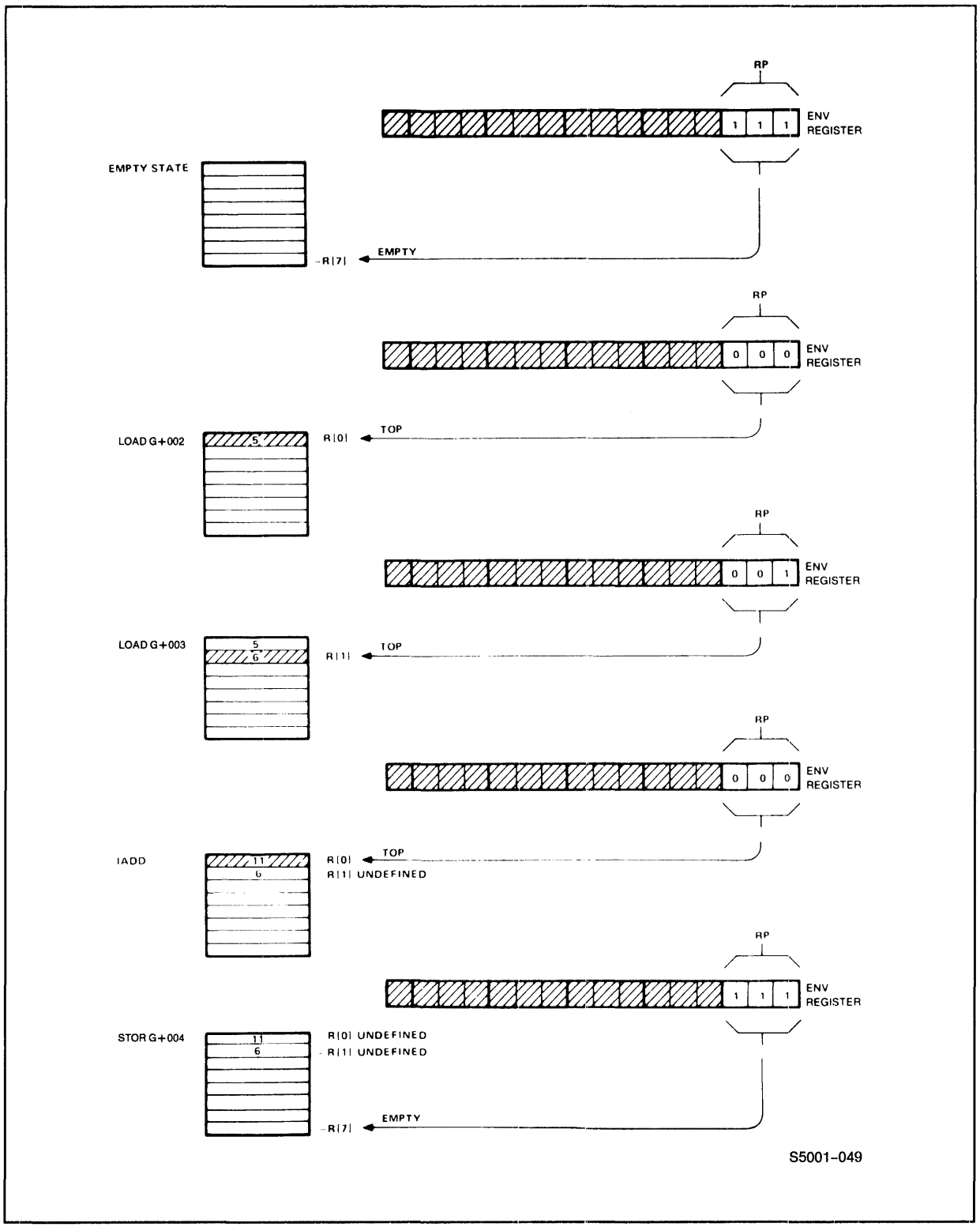


Figure 4-17. Action of the Register Pointer

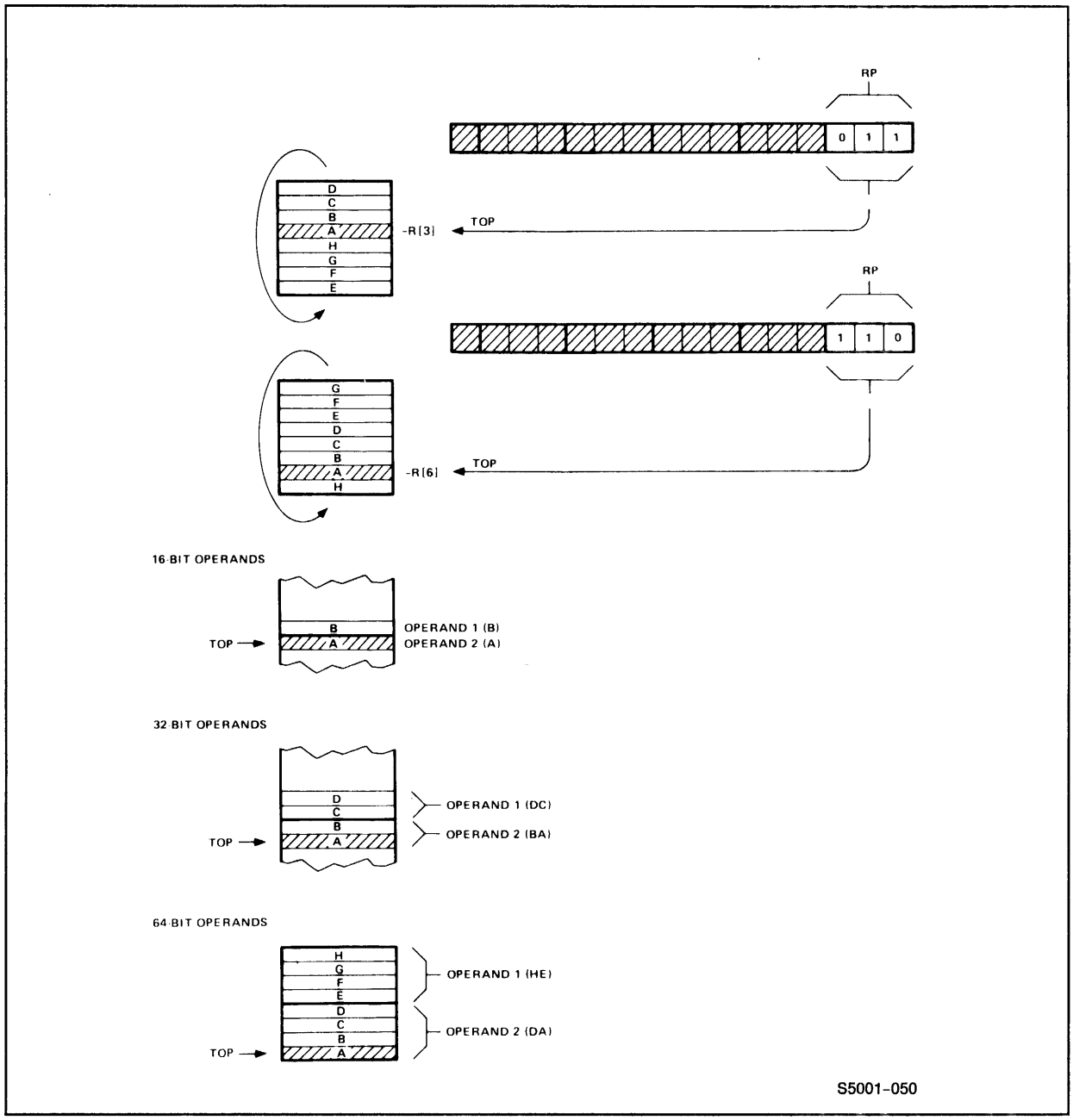


Figure 4-18. Naming Registers in the Register Stack

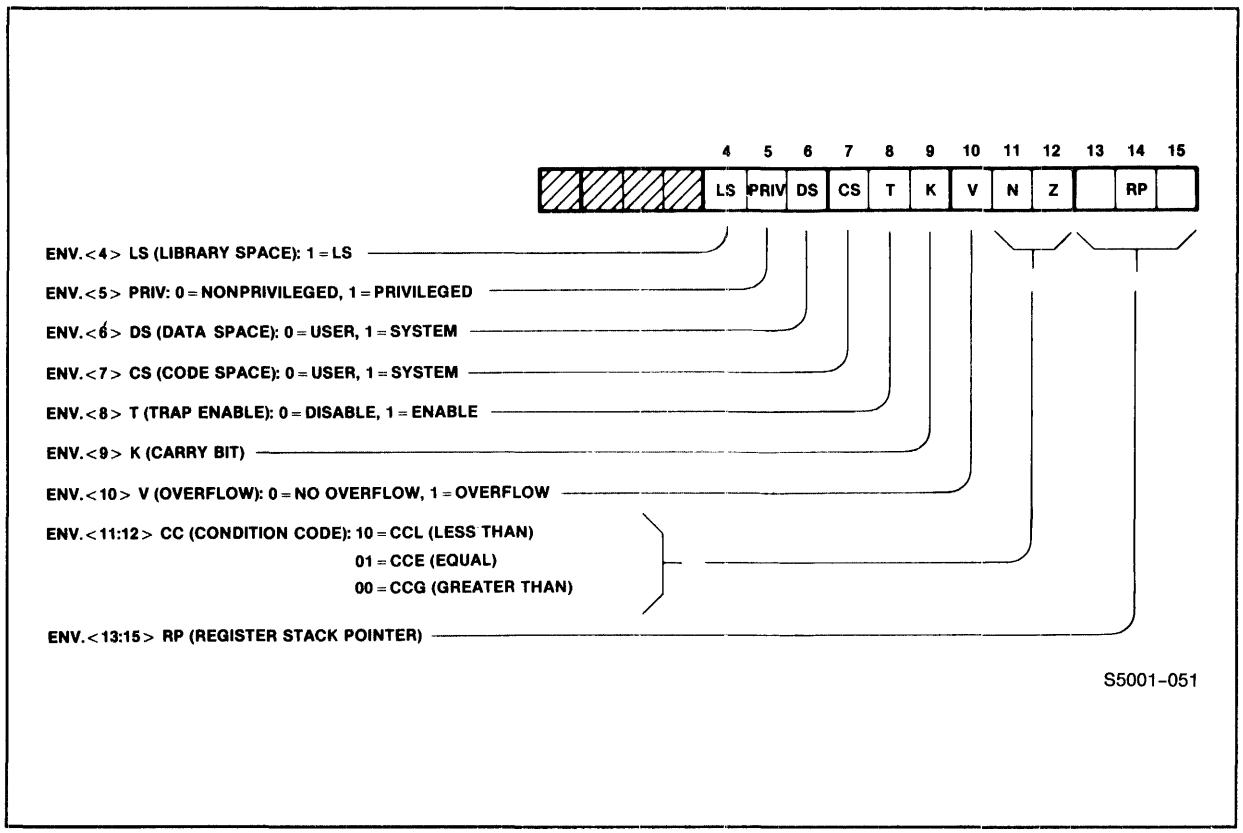


Figure 4-19. Environment Register

Library Space Bit. The LS bit (ENV.<4>) works with the CS bit (7) to define the current code space. When this bit is equal to 1, one of the library code spaces (user library or system library) is chosen for execution, rather than one of the standard code spaces (system code or user code), as selected by the CS bit. In the case of "system" selection by CS, the current system library segment is chosen for execution; in the case of "user" selection by CS, the user's current library segment is chosen for execution. (There can be up to 32 system library segments and up to 16 user library segments; only one of each is "current" at a given instant.)

Privileged Mode Bit. The PRIV bit (ENV.<5>), when equal to 1, means that the program is currently executing in privileged mode and is permitted to perform privileged operations. Privileged operations have the potential to adversely affect the operating system if misused. Some examples of privileged operations are: sending data over an interprocessor bus (SEND), initiating

input-output operations (EIO), calling privileged procedures, and accessing system tables. Normally, only the operating system executes in privileged mode; privileged operations are performed on behalf of application programs by the operating system. Nonprivileged programs can perform privileged operations only indirectly, by calling procedures designated callable. (Callable procedures execute in privileged mode but can be called by nonprivileged procedures.) When a nonprivileged procedure calls a callable procedure, its nonprivileged state is restored on return.

Instructions designated privileged can be executed only if the PRIV bit in the ENV register is equal to 1. If a nonprivileged program (i.e., PRIV = 0) attempts to execute a privileged instruction or call a privileged procedure, the firmware transfers control to the operating system instruction failure trap handler.

Data Space Bit. The DS bit (ENV.<6>) defines the current data segment. This specifies which data area is to be accessed when a data reference is made. DS, when equal to 0, specifies the user data segment; when equal to 1, it specifies the system data segment. DS equals 1 only in the interrupt environment; thus this bit is useful to both software and firmware in determining whether the current environment is an interrupt or a process. Processes executing in privileged mode can make explicit system data references regardless of the state of the DS bit through use of the SG-relative addressing mode.

Code Space Bit. The CS bit (ENV.<7>), together with the LS bit (ENV.<4>), defines the current code space. (Microcode selects the current segment within that space.) CS, when equal to 0, specifies the user code space (or the user library space if LS is equal to 1); CS equal to 1 specifies the system code segment (or the system library space if LS is equal to 1).

Trap Enable Bit. The T bit (ENV.<8>) specifies whether control is to be transferred to the operating system if an arithmetic overflow occurs or a divide with a divisor of 0 is attempted. If T is equal to 1 and an arithmetic overflow occurs (V, ENV.<10>, = 1), control is transferred to the operating system arithmetic overflow interrupt handler (see the GUARDIAN Operating System Programmer's Guide for possible recovery procedures). If T is equal to 0, control remains with the program having the overflow condition.

Generally, the T bit is under control of the operating system. However, application programs can set T to 0 by means of the SETE instruction if it is desired to handle arithmetic overflow conditions locally.

Carry Bit. The K bit (ENV.<9>), when equal to 1, indicates that a carry out of the high-order bit position occurred when executing an arithmetic instruction on a 16-, 32-, or 64-bit operand. The state of the K bit reflects the last arithmetic type instruction executed. The state of the K bit is also altered as the result of executing a scan instruction (SBW or SBU).

Two instructions test the state of the carry bit. They are:

BIC	Branch if carry
BNOC	Branch if no carry

Overflow Bit. The V bit (ENV.<10>), if equal to 1, indicates that an overflow condition occurred, or a divide (IDIV) with a divisor of zero was attempted. Overflow is generally associated with arithmetic operations on 16-, 32-, and 64-bit operands. Overflow also occurs in an LDIV instruction if the quotient cannot be represented in 16 bits, or in floating-point arithmetic if the exponent is too large or too small (see "Number Representation" in Section 3).

The state of the V bit is tested by the BNOV (Branch if No Overflow) instruction.

Condition Code Bits. This two-bit field (ENV.<11:12>) forms the Condition Code. The Condition Code generally reflects the outcome of a computation, comparison, bus transfer, or input-output operation. The Condition Code is also set by various system procedures to reflect the outcome of calls to those procedures, and by "load" instructions to identify the characteristics of the word or byte loaded onto the Register Stack.

The two bits that form the Condition Code are designated:

N = negative or numeric, ENV.<11>

Z = zero or alphabetic, ENV.<12>

The Condition Code has three states:

```
CCL = less than,    ENV.<11:12> = 10  (N = 1, Z = 0)
CCE = equal to,     ENV.<11:12> = 01  (N = 0, Z = 1)
CCG = greater than, ENV.<11:12> = 00  (N = 0, Z = 0)
```

The state of the Condition Code is tested by the following branch instructions:

BLSS	Branch if CCL	BLEQ	Branch if CCL or CCE
BEQL	Branch if CCE	BLEG	Branch if CCL or CCG
BGTR	Branch if CCG	BGEQ	Branch if CCE of CCG

The Condition Code is set explicitly by the following instructions:

CCL	Set CCL
CCE	Set CCE
CCG	Set CCG

The following paragraphs define the manner of setting the Condition Code in various cases.

Following a Computation. In this case, a hardware operation sets the Condition Code as follows, where x is the result of the computation:

```
CCL:   x  <  0
CCE:   x  =  0
CCG:   x  >  0
```

Following a computation, the Condition Code reflects the resultant value in a data segment location, on the top of the Register Stack, or in an index register. The location reflected by the Condition Code depends on the last instruction executed (see Section 9 for particulars). For example, a simple program to add two numbers and then store the result affects the Condition Code as follows:

```
Data in Global Area
   G [2]  =   5
   G [3]  =  -5

LOAD G + 002
   sets Condition Code to CCG (5 on the top of the Register
   Stack).

LOAD G + 003
   sets Condition Code to CCL (-5 on the top of the Register
   Stack).
```

IADD
sets Condition Code to CCE (0 on the top of the Register Stack).

STOR G + 004
does not change the Condition Code.

For a Comparison. In this case, a hardware operation sets the Condition Code bits as follows, where x and y are the operands:

For Signed Operands	For Unsigned Operands
CCL: x < y	CCL: x '<' y
CCE: x = y	CCE: x = y
CCG: x > y	CCG: x '>' y

The operand x is the first operand loaded onto the Register Stack (i.e., the second operand from the top of the stack), and y is the top operand in the Register Stack. For the DCMP instruction, x and y each take two registers on the Register Stack; for ECMP, each operand takes four registers. When two arrays are compared by a COMW or COMB instruction, x is the element in the destination array, and y is the element in the source array. For these instructions, the Register Stack is loaded with the address of the destination array (first item loaded), followed by the address of the source array (second item loaded) and the number of words or bytes to be compared (top of Register Stack). The single quote marks surrounding an operator symbol signify an unsigned (logical) rather than a signed (arithmetic) operation; thus '>' and '<' are unsigned comparison operators.

For a Byte Test. In this case, a hardware operation sets the Condition Code bits as follows, where x is the operand:

CCL: x is an ASCII numeric character
CCE: x is an ASCII alphabetic character
CCG: x is an ASCII special character

For a byte test, the Condition Code is set according to bits <8:15> of the operand on the top of the Register Stack when a BTST (Byte Test) or any "load byte" instruction (LDB, LBP, LBA, LBAS, LBX, LBXX) is executed. A Condition Code of CCL indicates that an ASCII numeric character (i.e., 0, 1, ..., 9) is on the top of the Register Stack. CCE indicates a lowercase or uppercase ASCII alphabetic character (i.e., a, b, ..., z or A, B, ..., Z), and CCG indicates an ASCII special character (i.e., neither numeric nor alphabetic).

For Bus Communication and Input-Output. For the Condition Code settings resulting from interprocessor bus communication, see the interprocessor bus description later in this section, and see the description of the SEND instruction in Section 9, "Instruction Set."

For input-output, see the input-output channel description in Section 7 and the EIO, IIO, and HIIO instructions in Section 9.

Register Stack Pointer. This three-bit field (ENV.<13:15>) defines the current top element of the Register Stack. The value of RP is implicitly changed by instructions that operate on values on the top of the Register Stack. RP is incremented as instructions are executed to load operands onto the Register Stack, and decremented when computations are performed or results stored.

The STRP instruction is used to explicitly set the RP value.

Environment Register Initial Settings. The ENV register is given an initial setting following a cold load to distinguish processor type. These settings are:

%3447 for a NonStop II processor. This setting specifies privileged mode, system data, system code, traps disabled, no carry, overflow, CCG, and RP = 7.

%3507 for a NonStop TXP processor. This setting specifies privileged mode, system data, system code, traps disabled, carry, no overflow, CCG, and RP = 7.

The ENV register is given the following setting whenever an interrupt handler is entered:

%3447 for a NonStop II processor
%3507 for a NonStop TXP processor

SETE Instruction. The SETE instruction is used to alter the ENV register contents. The bits of ENV.<8:15> can be set to any value desired; the bits of ENV.<0:7> are either cleared or left unchanged. This prevents nonprivileged processes from becoming privileged or gaining access to system data. A similar mechanism is used in the EXIT instruction to restore the ENV register contents when a procedure finishes. The programmer should take care when clearing ENV.<0:7>, since it is possible to inadvertently clear the Library Space (LS) bit, ENV.<4>.

PROCEDURES AND THE MEMORY STACK

A procedure is a functional block of instructions that, when called into execution, performs a specific operation. A procedure can perform an operation as simple as adding two numbers or as complex as locating an entry in a data base. A program typically consists of many procedures.

Several characteristics of procedures are:

- A procedure can be called into execution (invoked) from any point in a program.
- Procedures are assigned a "callability" attribute. The attribute specifies whether or not the caller must be executing in privileged mode, and whether or not the called procedure executes in privileged mode.
- The caller need not be concerned with its environment or the environment of the procedure it called, because:
  - --The caller's environment is automatically saved by the hardware when a procedure is called and is restored by the hardware when the called procedure finishes.
  - --When a procedure is called into execution, it is allocated its own temporary storage area called a local data area. The local data area (shown earlier in Figure 4-7) is known only to the executing procedure and is logically separate from other procedures' local data areas.
- Parameters (or arguments) can be passed to a procedure for evaluation. The parameters can be actual operands or can be addresses of operands.
- A procedure can return a value (such as the result of a computation) to its caller.
- A procedure itself can contain one or more subprocedures. A subprocedure is similar to a procedure in that it is also a functional block of instructions, called into execution to perform a specific operation. There are several similarities between procedures and subprocedures: a subprocedure, like a procedure, is allocated a temporary (sublocal) storage area while it executes, parameters can be passed to a subprocedure, and a subprocedure can return a value to its caller. Some significant differences between procedures and subprocedures are: different instructions are used to call a subprocedure than a procedure, a subprocedure has no "callability" attribute (it executes in the mode of its caller), and the amount of sublocal storage available to a subprocedure is

significantly less than the amount of local storage available to a procedure. (BSUB and RSUB do not change the current L or ENV register setting.) In addition, a subprocedure can be called only by the procedure that contains it, or by another subprocedure contained within the procedure. A subprocedure can access both its sublocal storage as well as the procedure's local and global storage.

A procedure consists of a contiguous block of instruction codes and program constants in a code segment. The procedures that compose a program are in one or more segments within the user code space. They may call procedures in the system code segment, any of the system library segments, or any of the segments in the user library space. The address of the first instruction in a procedure is called the entry point. The entry points for all procedures in a program are located in a table, known to the hardware, called the Procedure Entry Point (PEP) table. The PEP itself is located at the beginning of each code segment. See Figure 4-20.

The External Entry Point table, also shown in Figure 4-20, exists in each segment, but will be discussed later under "Calling External Procedures." This table ends on a page boundary, with entries consecutively assigned backward toward the end of code, using the first available space that fits (either on the same page as the end of code or on a separate page).

Procedures are invoked using procedure call instructions--PCAL to a procedure within the same code segment, or XCAL to a procedure in some other code segment. During execution of either of these instructions, the caller's environment (specifically, the address of the instruction following the call, the L register setting, and the current ENV register setting--modified to include space ID index) is saved in a three-word stack marker. The stack marker is written at the current top of the memory stack. The call instruction then references the entry in the PEP table corresponding to the procedure being called. The address in the PEP entry is placed in the P register so that the next instruction executed is the one at the procedure's entry point.

The last instruction that a procedure executes is an EXIT instruction. The EXIT instruction is used to return control to the caller. Specifically, the caller's L register setting is restored, and the return address (i.e., that of the instruction following the call instruction) is set into the P register. The caller's ENV register setting also is restored--except for the Condition Code (CC) and Register Pointer (RP) fields, which are left as is, since these fields in the stack marker copy of ENV were used to save the space ID index. The EXIT instruction microcode performs a segment switch, using the space ID, if the caller's segment is different from the segment of the called procedure.

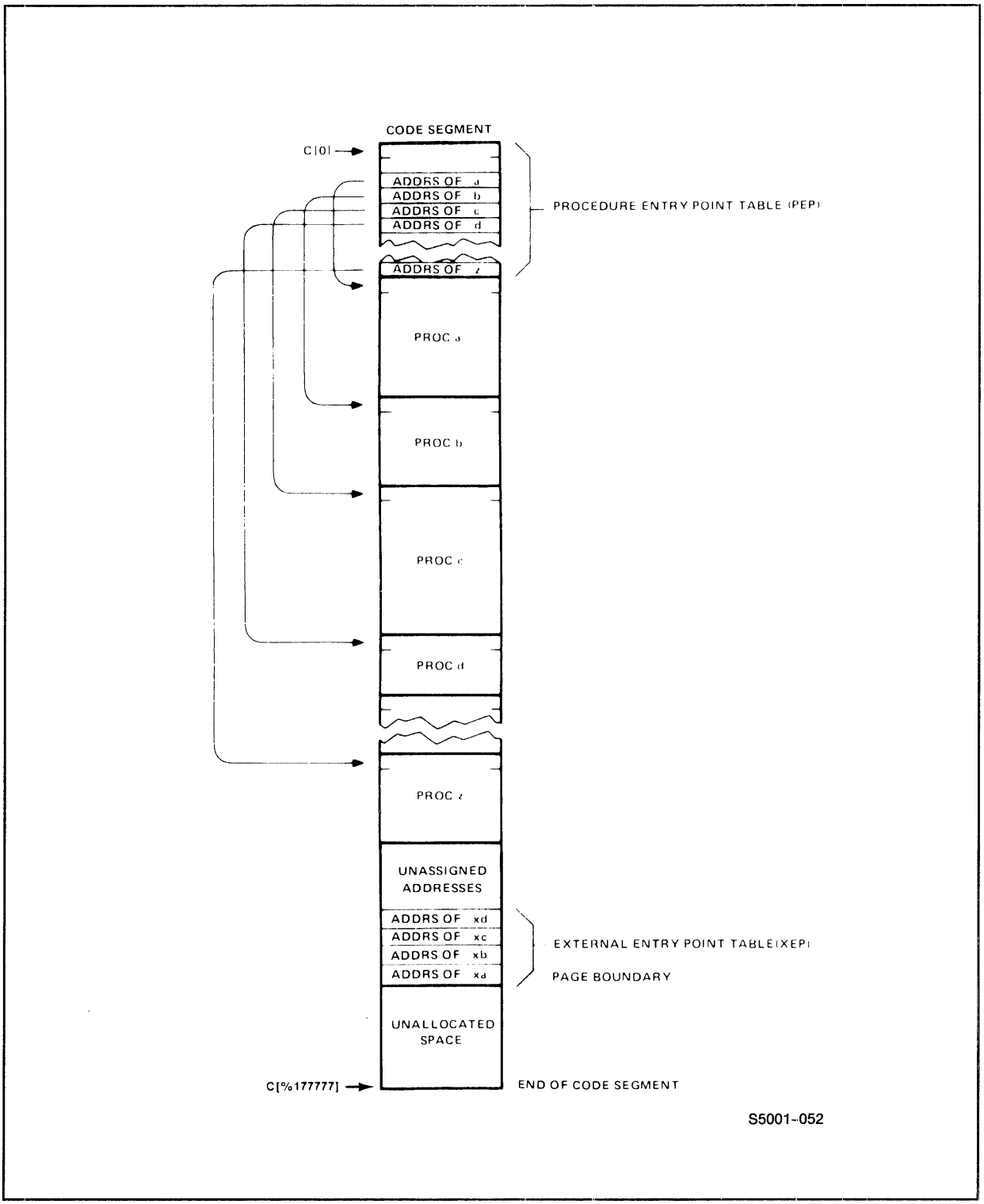


Figure 4-20. Procedure Entry Point and External Entry Point Tables

An example of a procedure call and exit is shown in Figure 4-21. This example assumes the called procedure is in the same segment as the caller, UC.2; note the space ID index value of 2 in the stack marker.

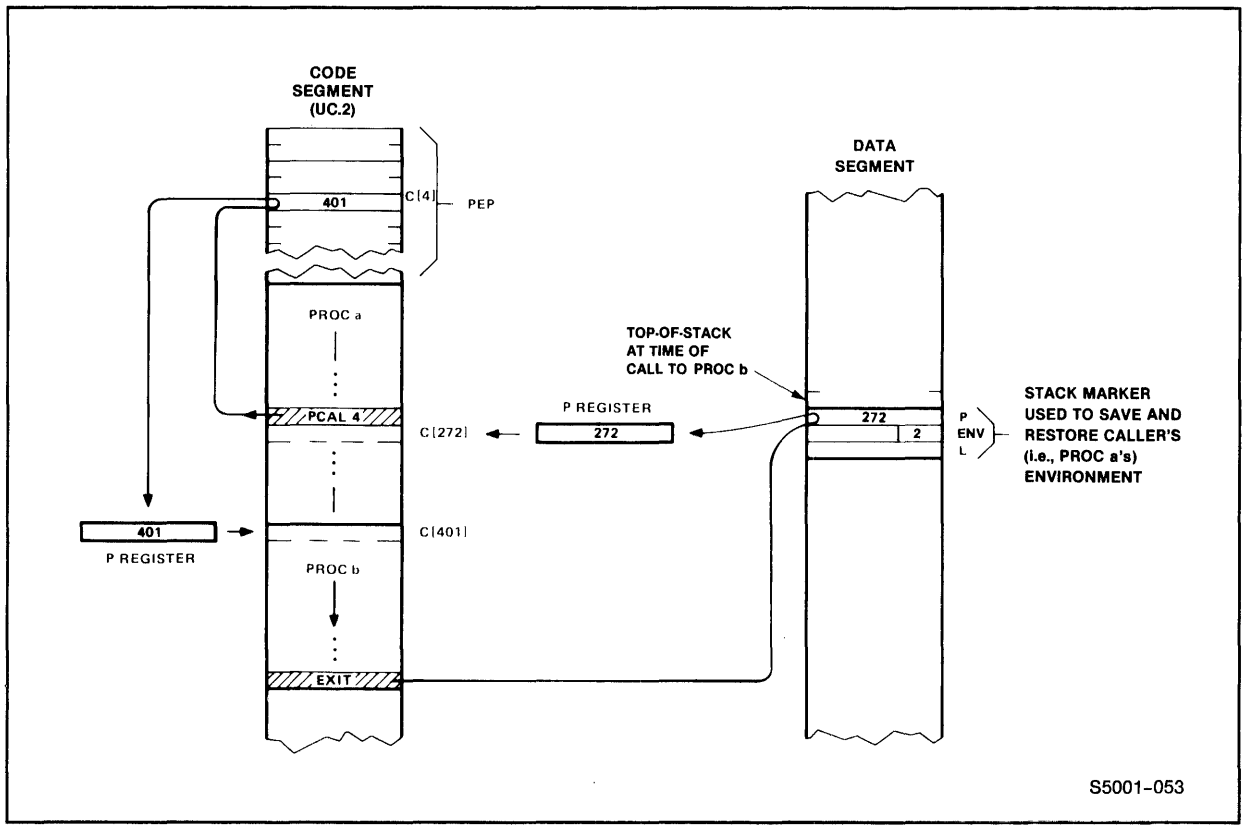


Figure 4-21. Procedure Call and Exit

Attributes of Procedures

So that a nonprivileged process cannot execute in privileged mode and so that execution of privileged operations can be controlled, every procedure has one of the following attributes:

- Nonprivileged. Procedures having this attribute can be called by any procedure. They execute in the same mode (privileged or nonprivileged) as the calling procedure. This is the attribute typically given to procedures in an application program.

- Callable. Procedures having this attribute can also be called by any procedure, but they execute in privileged mode (i.e., PRIV = 1). The caller's mode is restored when a callable procedure exits. This attribute is typically assigned only to operating system procedures. It is used so that a controlled interface exists between a nonprivileged application program and the privileged operating system.

- Privileged. Privileged procedures execute in privileged mode and are callable only by procedures currently executing in privileged mode. An attempt by a nonprivileged procedure to call a privileged procedure results in an illegal instruction trap. This attribute should be used only by the operating system. It is typically used when an operation, if done improperly, might have an adverse effect on processor module operation. A nonprivileged application program's only interface to an operating system privileged procedure is through a procedure with the callable attribute. (For example, many of the GUARDIAN file system procedures described in the System Procedure Calls Reference Manual are callable procedures that, in turn, call privileged operating system procedures.)

In the PEP table, procedure entry points are grouped according to attribute. There are three groups: the first is nonprivileged procedures, the second is callable procedures, and the last is privileged procedures.

The first two words in the PEP table, C[0:1], describe where the callable and privileged entry points begin in the PEP. Specifically, C[0] is the address of the first PEP entry for a callable procedure, and C[1] is the address of the first PEP entry for a privileged procedure. See Figure 4-22. These words are used to check whether a nonprivileged caller is attempting to invoke a privileged procedure.

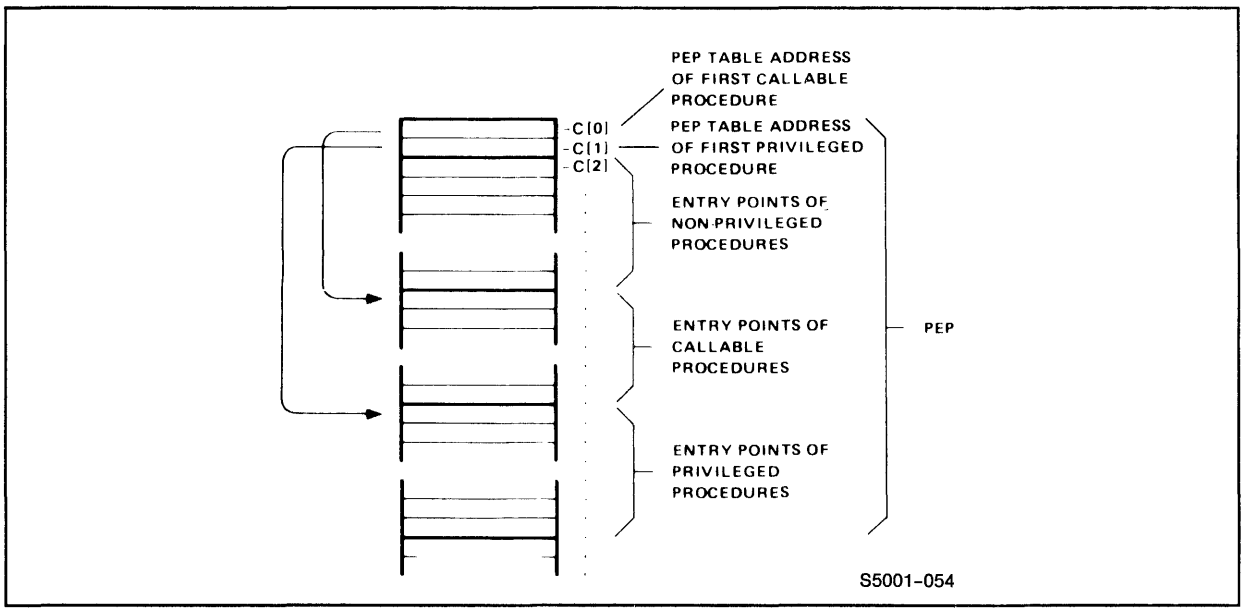


Figure 4-22. First Entries in Procedure Entry Point Table

PCAL Instruction

The steps involved when a Procedure Call (PCAL) instruction is executed are described below, with step numbers referring to the accompanying illustration, Figure 4-23. Note that before the PCAL executes, the procedure parameters (and the mask word or words, for procedures with a variable number of parameters) must be pushed onto the stack. Also, it is usually assumed by the called procedure that the Register Stack is empty when a PCAL is about to be executed (RP=7). The RP (Register Pointer) and CC (Condition Code) fields of ENV that are saved in the stack marker (step 1, below) are overwritten by the space ID index of the calling procedure.

1. The caller's environment is saved in a three-word stack marker.

```
data [S+1] := P;     !
data [S+2] := ENV;   ! stack marker.
data [S+3] := L;     !
```

   The stack marker is pushed onto the top-of-stack location, as indicated by the address in the S register. The stack marker contains the following information:

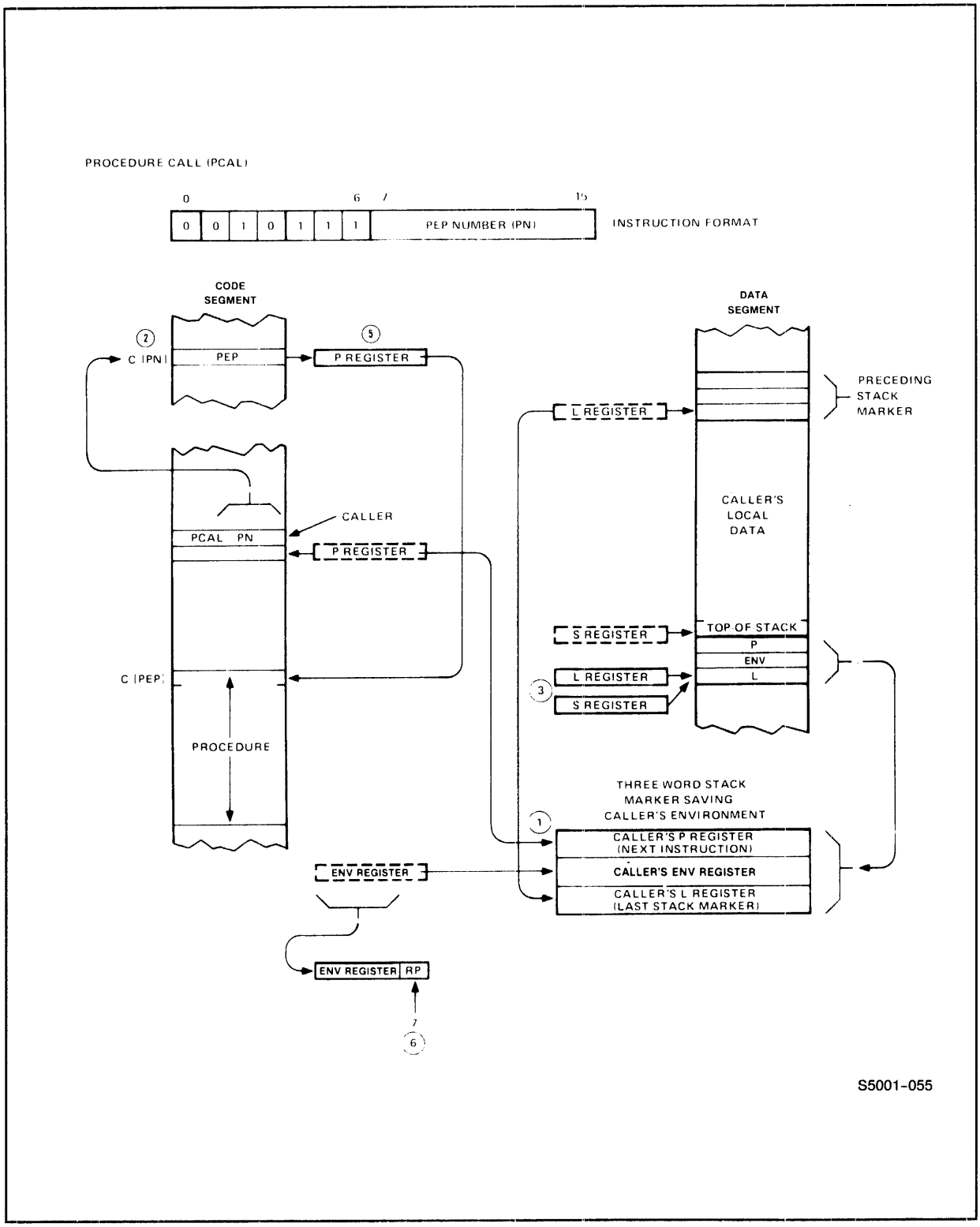


Figure 4-23. Execution of PCAL Instruction

- the current P register setting (the address of the instruction following the PCAL)
- the current ENV register setting
- the current L register setting (the beginning of the caller's local data area)

NOTE

The stored copy of the ENV register (see Figure 4-24) contains the complete space identification of the caller's segment, since code spaces can have multiple segments. This consists of the LS and CS bits, to select one of the four code spaces, plus a space ID index to select a specific segment within that code space. (Also note that, since bits 11 through 15 of ENV, which normally contain CC and RP, are used to save the space ID index in the stack marker, the CC and RP fields must not be modified in the stack marker.)

2. If the calling procedure is not executing in privileged mode, the "callability" attribute of the procedure being called is checked.

   First, the PEP number field of the PCAL instruction is compared with the entry in C[0] (the address of the first PEP entry for callable procedures). If the PEP number is greater than or equal to the C[0] entry, then this is a call to a callable or privileged procedure, so a second check is made: the PEP number field of the PCAL instruction is compared with the entry in C[1] (the address of the first PEP entry for privileged procedures). If the PEP number is greater than or equal to the entry in C[1], then this is a call to a privileged procedure; so, an instruction failure trap occurs,

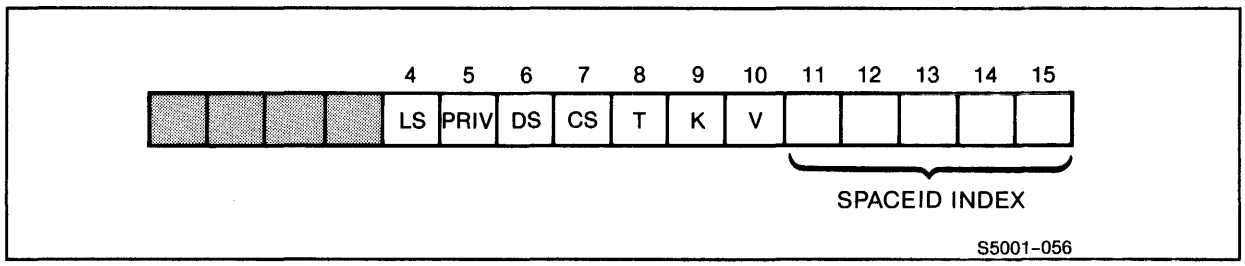


Figure 4-24. Space Identification in Stored Copy of ENV

and the PCAL instruction fails. Otherwise, this is a call to a callable procedure, so the PRIV bit (in ENV) is set.

If the PEP number is less than the C[0] entry, then this is a call to a nonprivileged procedure, so no special action is taken.

3. The S and L registers are set with the G[0]-relative address of the new top-of-stack location (the third word of the stack marker).

```
L := S := S+3;
```

The new L register setting defines the base of the local area for the procedure being called.

4. The new S register setting is tested for an address within the memory stack area, G[0:32767]. If the value is greater than 32,767, control is transferred to the operating system stack overflow trap (and the PCAL instruction is aborted).

```
if S '>' 32767 then stack^overflow^trap;
```

5. The C[0]-relative address of the procedure being called is obtained from the PEP table entry pointed to by the <PEP number> field in the PCAL instruction. This address is put in the P register so that the next instruction executed will be the first instruction of the called procedure.

6. Finally, RP is given an initial value of seven (stack empty) if it does not already have this value.

```
RP :=  7;
```

Following the PCAL, the instructions comprising the procedure are executed. The last instruction that a procedure executes is an EXIT instruction.

EXIT Instruction

The EXIT instruction uses the three-word stack marker to restore the caller's environment. The sequence is as follows, with reference to Figure 4-25. (For simplicity, and continuity with

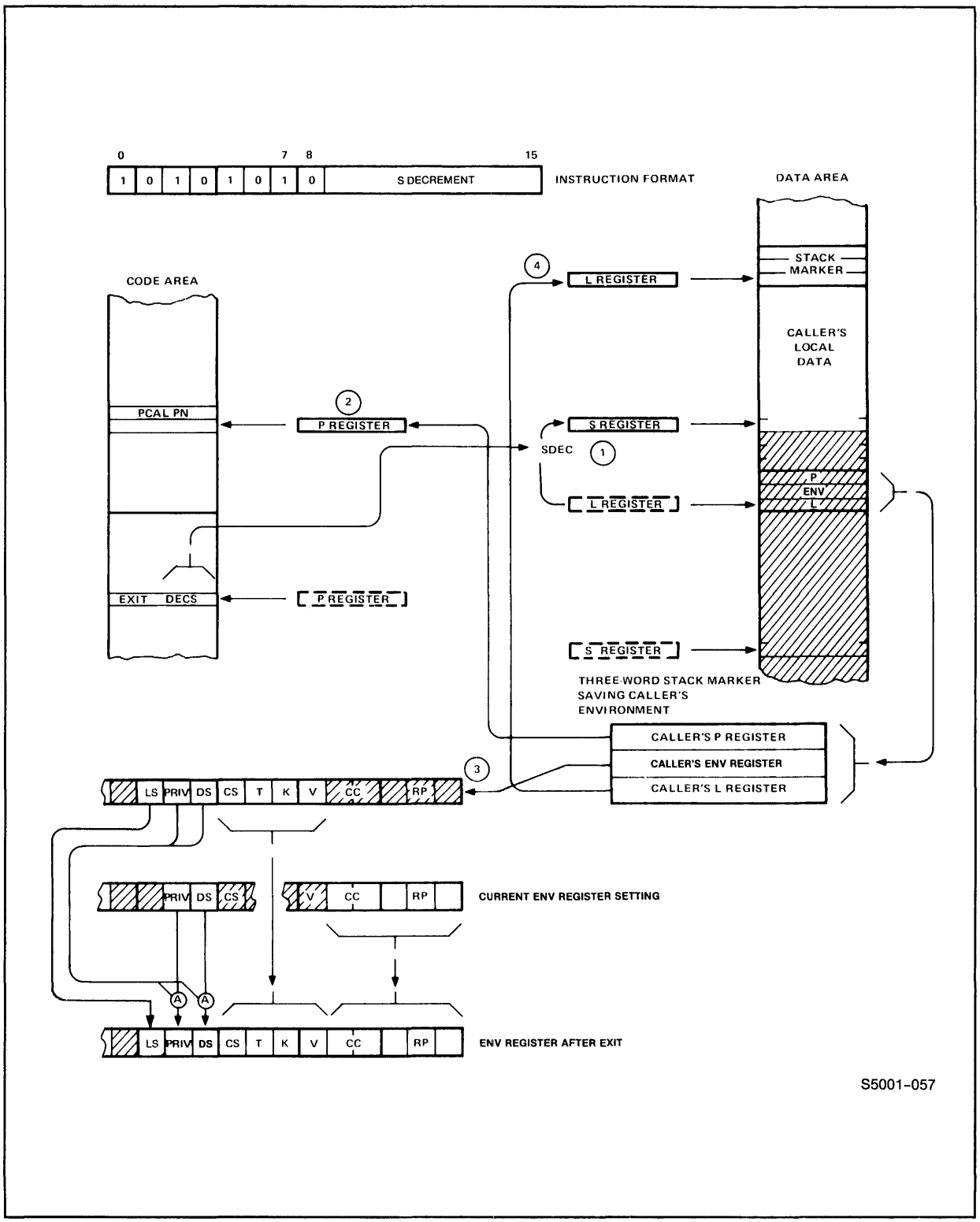


Figure 4-25. Execution of EXIT Instruction

the preceding PCAL description, this sequence assumes the return is from a procedure that was called with PCAL rather than XCAL.)

1. The S register setting is moved below the local area, the stack marker, and any parameters to the exiting procedure.

   ```
   S := L - S^decrement;
   ```

   The "S^decrement" value (which is specified in the EXIT instruction) is subtracted from the current L register setting and placed in the S register. The value of "S^decrement" is three (for the stack marker) plus the number of words of parameter and mask information passed to the exiting procedure.

2. The P register is set with the P register value saved in the stack marker at L[-2].

   ```
   P := data [L-2];
   ```

   The next instruction to be executed will be the one following the PCAL instruction.

3. The ENV register is restored from a combination of the current ENV register setting and the ENV register value saved in the Register Stack at L[-1].

   The mode (privileged or nonprivileged) and data area are reestablished to be the lesser of the caller's and the current settings. This ensures that a nonprivileged user cannot exit with privileged capability. The caller's CS (code space), LS (library space), T (traps), V (overflow), and K (carry) are reestablished from L[-1]. Z and N (Condition Code) are left at their current settings to reflect the results of the call. RP is left at its current setting so that a value in the Register Stack can be returned to the caller.

4. The L register is restored from the L register value saved in the stack marker at L[0].

   ```
   L := data [L];
   ```

   This moves L back to point to the preceding stack marker, thereby reestablishing the preceding local data area.

The instruction following the PCAL instruction then executes.

CALLING EXTERNAL PROCEDURES

Procedures in an external code segment can be called almost as efficiently as the current segment's own procedures. The XCAL (external procedure call) instruction and the space ID addressing convention are two important features that make this possible.

Figure 4-26 illustrates an example of a call from a user code segment to a procedure in the system code segment. (The general method applies to any allowable external call between any pair of segments in any of the four code spaces--user code, user library, system code, and system library.) When the application program program calls the external procedure, an XCAL instruction is executed. This instruction places a three-word stack marker on the top of the user stack and moves L and S in the same manner as a PCAL instruction (i.e., defines a new local area). However, instead of transferring control directly to a procedure within the segment, control is vectored out of the segment (via its XEP, External Entry Point table) into another code segment (through that segment's PEP, Procedure Entry Point table). In this example, the system code segment's Procedure Entry Point table (PEP) is used to determine the procedure's starting address, and the CS bit in the ENV register is set to "1" so that instructions will be executed from the system code segment. The DS bit, however, remains a "0" so that the user data segment (as opposed to the system data segment) is still in effect. The local area for the system procedure is therefore in the user data segment. Specifically, the steps involved when the XCAL instruction is executed are:

1. The caller's environment is stored in a stack marker.

```
data [S+1] := P;
data [S+2] := ENV;
data [S+3] := L;
```

   The stored copy of the ENV register (see Figure 4-24) contains the complete space identification (LS and CS bits, plus the space ID index for the selected code space) of the caller's segment, since code spaces can have multiple segments. (Note that hardware bits 11 through 15 of the ENV register, which normally contain the Condition Code and Register Pointer, are not saved.)

2. The C[0]-relative address of the procedure being called is obtained by a three-step process. First, the XCAL instruction specifies a location in the caller's External Entry Point table (XEP; refer back to Figure 4-20). Then, the XEP entry is used to locate the desired code segment

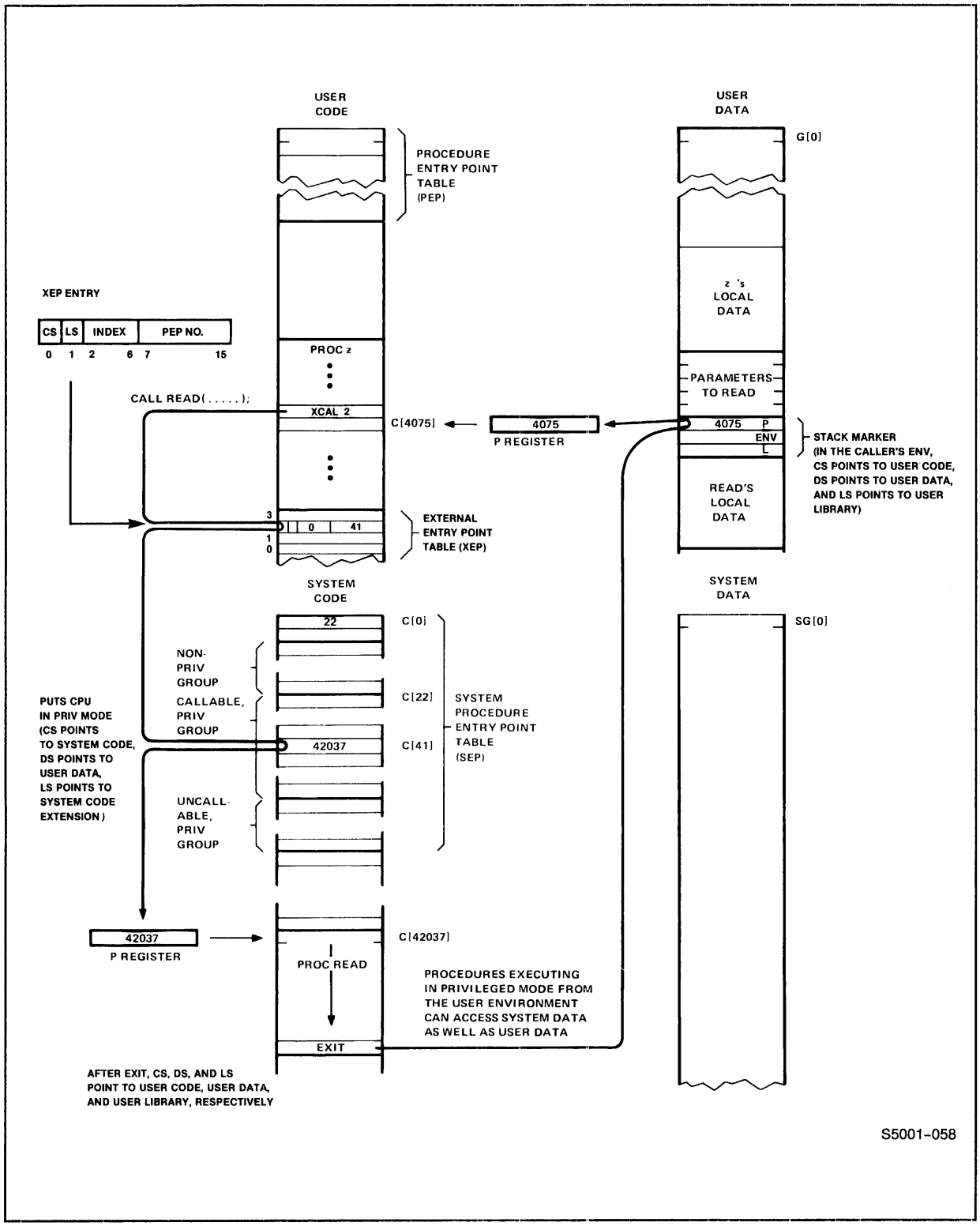


Figure 4-26. System Procedure Call and Exit

(bits 0 and 1 specify CS and LS respectively, and bits 2 through 6 specify the segment index) and a Procedure Entry Point number (bits 7 through 15 of the entry)--which in this case is in the system code segment's Procedure Entry Point table. Finally, the address in that PEP entry is put in the P register so that the next instruction executed will be the first instruction of the system procedure.

3. If the calling procedure is not executing in privileged mode, the callability attribute of the system procedure being called is checked.

```
sas := 3;    ! system code segment--in this case
temp := <PEP number>;
if not PRIV then
  if temp >= mem(3,0) then    ! call to callable
    begin
      if temp >= mem(3,1) then    ! call to privileged
        instruction^failure^trap;
      PRIV := 1; ! set privileged mode
    end;

P := mem(sas,temp);    ! get entry point address into P
```

4. The S and L registers are set with the G[0]-relative address of the new top-of-stack location.

```
L := S := S + 3;
```

The new L register setting defines the base of the local area for the system procedure being called.

5. The new S register setting is tested for an address within the memory stack area, G[0:32767]. If the value is greater than 32,767, control is transferred to the operating system stack overflow trap (and the XCAL instruction is aborted).

```
if S > 32767 then stack^overflow^trap;
```

6. The CS bit of the ENV register is set to 1 and the LS bit is set to 0, so that further code area references will be in the System Code segment (in this example). CS and LS settings are derived from bits 0 and 1 of the XEP table entry, respectively.

7. Finally, the Register Stack Pointer, RP, is given an initial value of seven (stack empty).

When the system procedure finishes, the EXIT instruction is executed. The CS and LS bits, plus the space ID index bits from the stored copy of the ENV register, are used to reestablish the caller's segment of the user code space as the currently selected code space, so that the next instruction is executed from that segment.

NOTE

The foregoing example of an external procedure call is the most straightforward case, in that the call is to the system code segment (SC.0), which is always fully mapped. If the call had been to a multisegment code space (system library, or user code or library), the possibility exists that the target segment might not have been currently mapped. In that case, the XCAL instruction automatically executes the MAPS instruction during step 2 above, before proceeding with the remaining steps. (The EXIT instruction similarly invokes MAPS when necessary, prior to any of the four steps shown earlier in Figure 4-25.)

MEMORY STACK OPERATION

Figures 4-27a and b depict an example of a memory stack operation from an initial state (i.e., start of process execution) through a call to, and subsequent return from, a procedure. The purpose of the diagram is to show the action of the L and S registers as a procedure generates its local variables and prepares to call a procedure by passing parameters, how L and S are set when a procedure is called, and how L and S are set when the return is made to the caller.

1. Initial State

   After the operating system has loaded a program into memory but before the first instruction of the process executes, the following initial conditions are present: the process's global variables are initialized and present, and the L and S registers are set to the address of the word just past the global area. There are no local variables defined at this time.

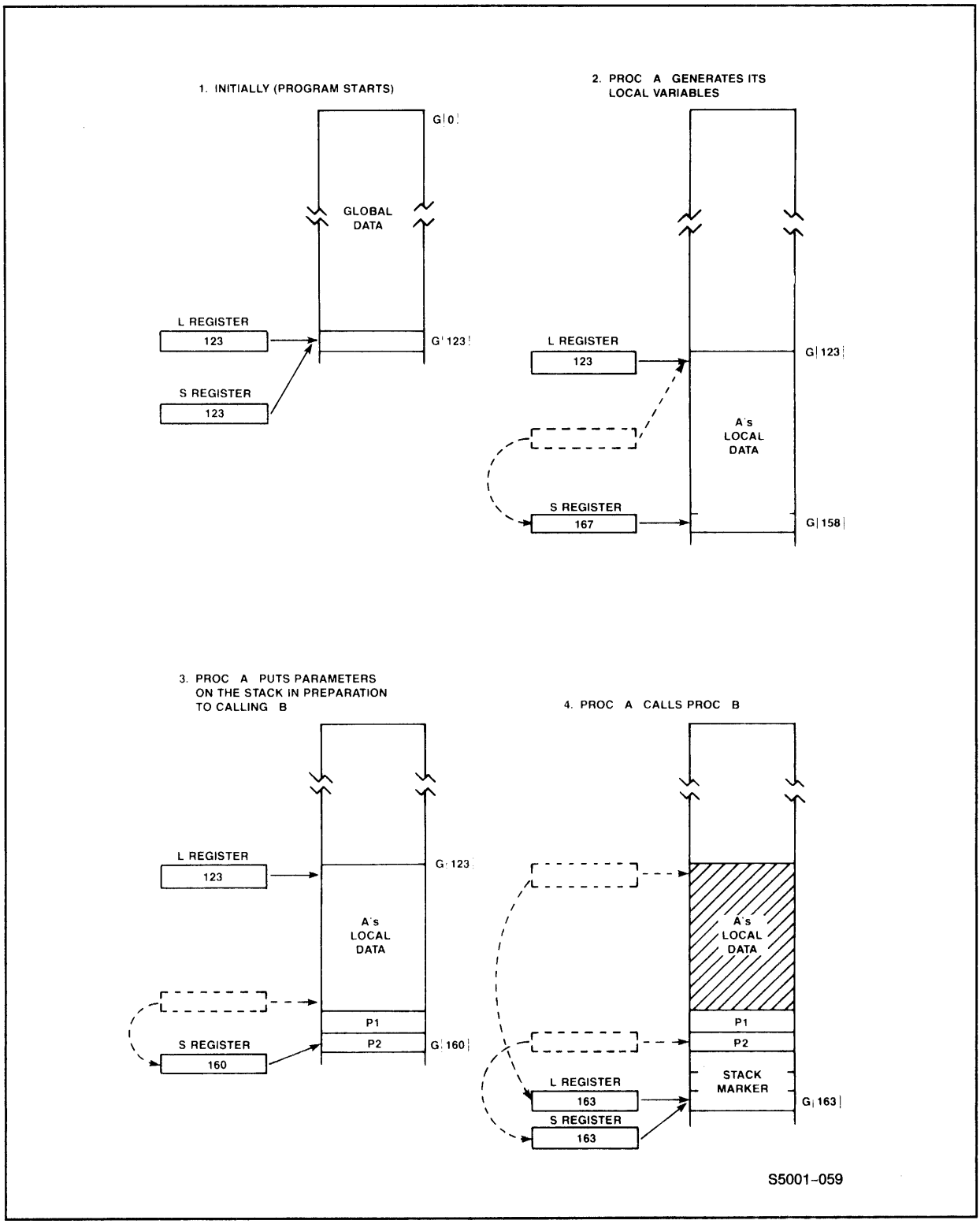


Figure 4-27a. L and S Registers in Procedure Calls

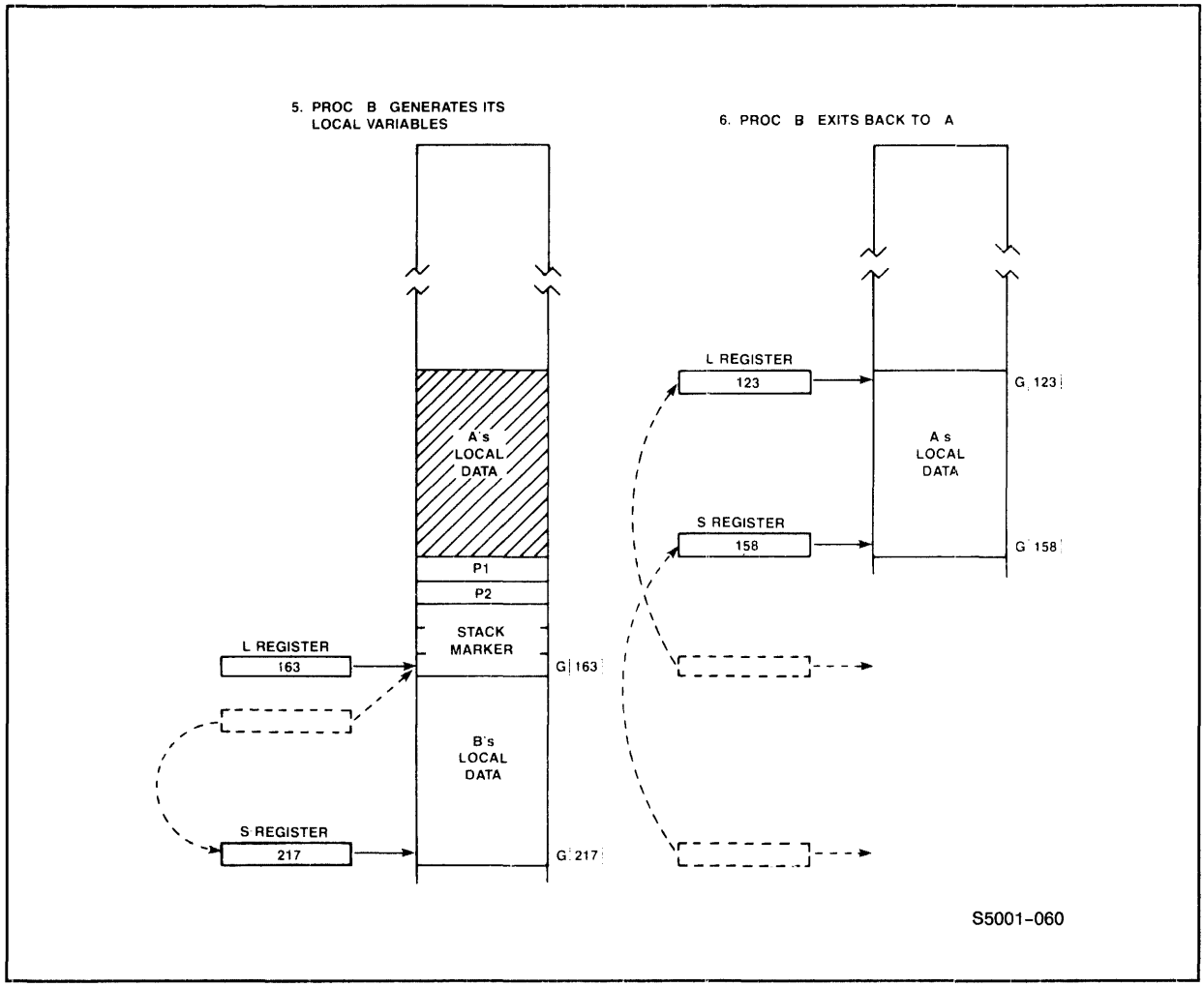


Figure 4-27b. L and S Registers in Procedure Calls

2. Procedure A generates its local variables

   The first few instructions of a procedure generate the procedure's local variables. As the local variables are generated, the S register setting increases, defining a new upper limit to the procedure's local area. Note that the L register setting does not change.

3. Procedure A passes parameters to procedure B

   In preparation for calling the procedure B, the parameter words (two in this example) are placed on the top-of-stack location as indicated by the S register setting. The S

register setting is increased by two to account for the parameters.

4. A calls B

After the parameters are loaded onto the memory stack, a procedure call instruction is executed; this could be a PCAL (Procedure Call), XCAL (External Procedure Call), or DPCL (Dynamic Procedure Call). Figure 4-27 assumes PCAL. Execution of the call instruction places a three-word stack marker at the current S register setting plus one (just above the parameters). L and S registers are given a new setting; they both point to the third word of the stack marker. The new L register setting defines the start of B's local area. At this point, no local variables have been generated for procedure B. (Note that A's local area, which is normally addressed relative to the L register, is no longer addressable by the L-plus addressing mode.)

5. Procedure B generates its local variables

In the same manner as procedure A did, procedure B generates its local variables. This increases the S register setting accordingly, so that the S register defines the new upper limit to B's local area.

6. Procedure B exits back to procedure A

When procedure B completes, an EXIT instruction is executed to return to A. Execution of the EXIT instruction moves the L register setting back to the beginning of A's local area and moves the S register setting back to the top-of-stack location that was in effect before the parameters were loaded on the stack (this is accomplished by the "S^decrement" value in the EXIT instruction). Specifically, for the return to procedure A, the EXIT instruction is:

```
EXIT 5
```

This deletes the three-word stack marker, plus the two parameter words, from the top of the stack.

Generation of and Access to Local Data

Unlike the global data area, which exists at all times, the local data area for a procedure exists only while the procedure is actually executing. The local variables are generated and initialized by instructions at the start of a procedure's code. Thus, a procedure can be called any number of times (and in fact can call itself), and each call generates a fresh copy of the procedure's local data area. An example of the instructions used to generate the following local variables are considered next (referring to Figure 4-28):

```
INT i,              !  L[1]
    j := 5,         !  L[2]
    .k [0:31];      !  L[3]  (pointer to k, which starts at L[4])
```

These are three local variables declared in a TAL source program: "i" is a one-word uninitialized variable, "j" is a one-word variable initialized with the value 5, and "k" is an indirectly addressed array variable consisting of 32 words. The instructions to generate these variables are:

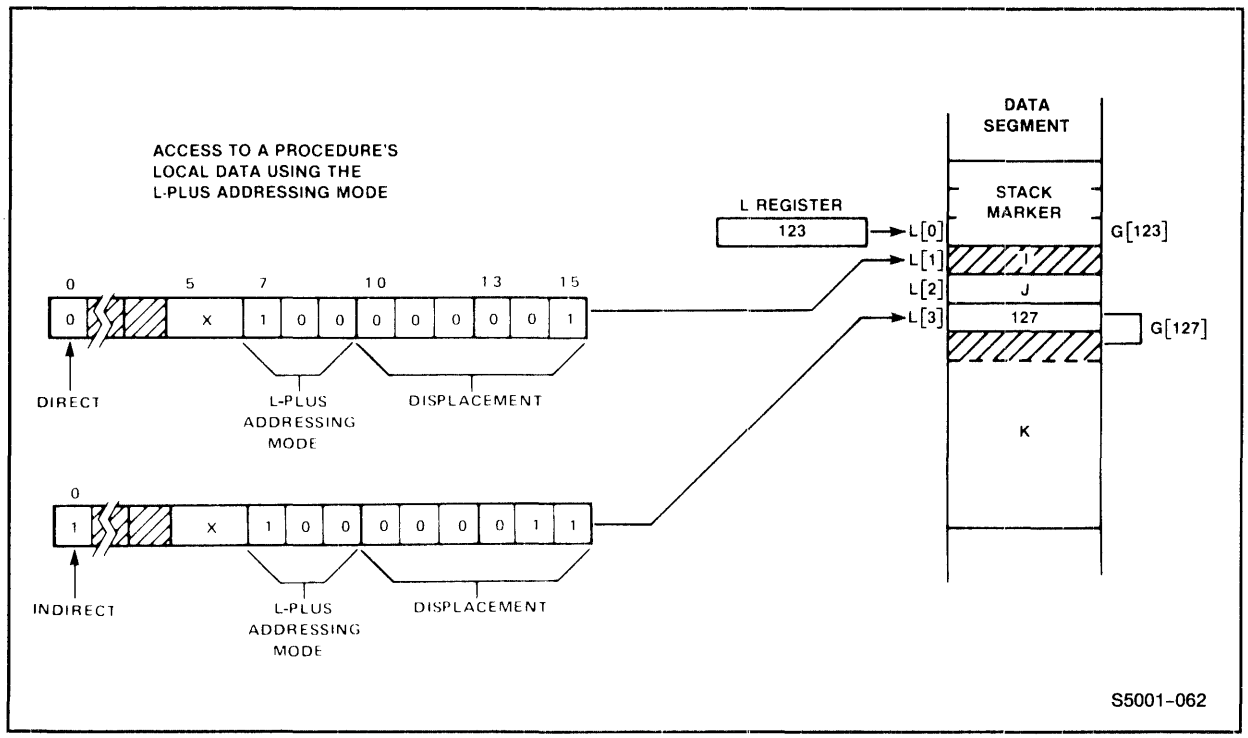


Figure 4-28. L-Plus Addressing Mode

```
ADDS     +001      ! Add to S
LDI      +005      ! Load Immediate
LADR    L+004      ! Load Address
PUSH     711       ! PUSH to Memory
ADDS     +040      ! Add to S
```

The ADDS instruction increments the S register setting by one. This allocates one word for the variable "i".

The LDI instruction puts the initialization value for "j" (5) on the top of the Register Stack.

The LADR instruction calculates the G[0]-relative address of the first word of the indirect array "k" and puts the address on the top of the Register Stack.

The PUSH instruction performs two functions: (1) it puts the initialization value given in "j" and the address of the array "k" into L[2] and L[3] of the process's stack, respectively, and (2) it increments the S register setting by two to allocate the two words needed for "j" and the address pointer to "k".

The ADDS instruction increments the S register setting by 32 (octal 40). This allocates 32 words for the indirect array "k".

Following the generation of the local variables, the local area for this example consists of:

```
L[1]    = i
L[2]    = j (initialized with a value of 5)
L[3]    = an address pointer to the array "k"
L[4:35] = the array "k"
```

Once allocated, data in the local area is addressed relative to the current L register setting using the L-plus addressing mode. As illustrated, this mode can access local data directly or can use the direct address as an address pointer (indexing is also permitted).

The top-of-stack area is addressable implicitly through use of the PUSH and POP instructions. These are illustrated in Figure 4-29. The PUSH instruction is used to store the Register Stack contents, usually prior to calling a procedure, on the top of the memory stack. When a PUSH instruction is executed, the S register setting is incremented by the number of words pushed. The POP instruction is used to restore the Register Stack contents from the top of the memory stack, then decrement the S register setting accordingly.

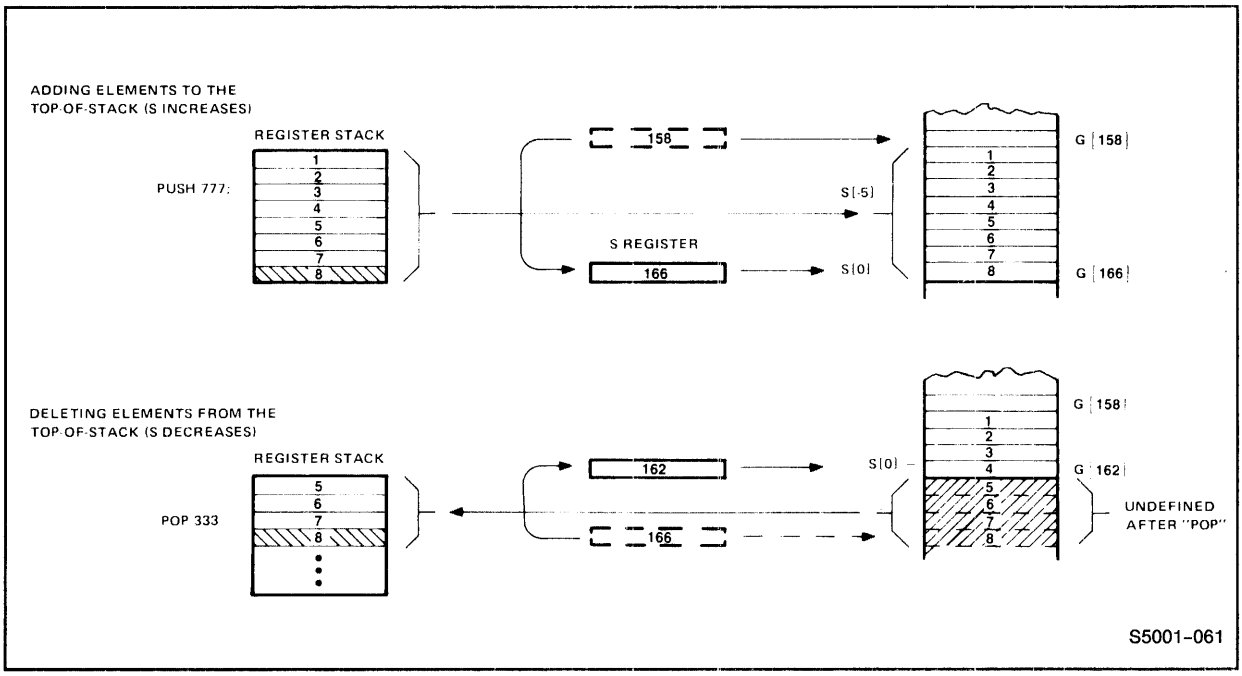


Figure 4-29. PUSH and POP Instructions

Parameter Passing

Parameters are passed to a procedure in the top-of-stack area. Naturally, there must be coordination between the caller and the called when passing parameters. The caller must know the order in which a procedure expects parameters, and whether a parameter is to be an actual operand (called a value parameter) or an address pointer (called a reference parameter).

Before the caller invokes a procedure, the parameters are prepared in the Register Stack. The actual operands (for value parameters) and the addresses of operands (for reference parameters) are loaded into the Register Stack in the order required by the procedure being called. The address of a reference parameter is obtained by the execution of a LADR (load address) instruction. The parameters that have been prepared in the Register Stack are loaded on the top of the memory stack by executing a PUSH instruction (which increments the S register accordingly).

An example will now be considered to show the instructions used to prepare the top of the memory stack area for parameter passing. This example uses the variables declared in the preceding example, and is illustrated in Figure 4-30. The procedure being called is of the form:

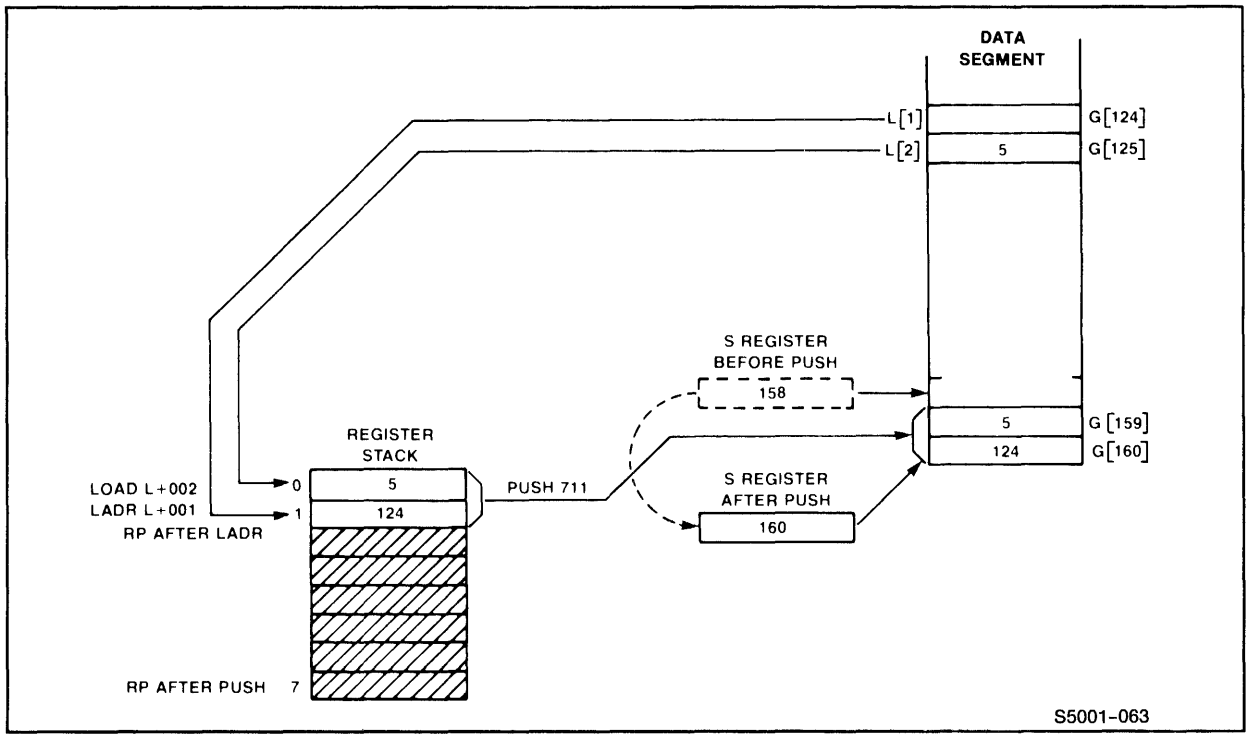


Figure 4-30. Parameter Passing

```
PROC b (p1,p2);
   INT p1,.p2;
```

Parameter "p1" is a value parameter; therefore, the procedure expects an actual value to be passed. Parameter "p2" is a reference parameter, and, therefore, the procedure expects the G[0]-relative address of a variable to be passed.

The call being made from procedure A is:

```
CALL b (j,i);
```

The instructions to pass these two parameters are:

```
LOAD L +002
LADR L +001
PUSH 711
```

The LOAD instruction puts the contents of the variable "j" (the value 5) on the top of the Register Stack. (This is the parameter passed as "p1", a value parameter, to procedure B.)

The LADR instruction calculates the G[0]-relative address of the variable "i" and puts the address on the top of the Register Stack. (This is the parameter passed as "p2", a reference parameter, to procedure B.)

The PUSH instruction places the two parameters from the Register Stack on the top of the memory stack and increments the S register setting by two.

Parameter Access

Parameters are accessed by using the L-minus addressing mode. This mode provides access to the 32 locations just below and including the current L register setting (L[-31:0]). Subtracting the three words used for the stack marker, this leaves 29 words addressable as parameters. If value parameters are passed, the parameter location is addressed directly (<i>, indirect, bit of a memory reference instruction = 0); if reference parameters are passed, the parameter location is used as an indirect address (<i> bit = 1). Indexing in either mode is permitted.

Figure 4-31 shows an example of both value and reference parameter access.

Returning a Value to the Caller

A procedure can return a value to its caller using the top of the Register Stack. This, like parameter passing, requires coordination between the caller and the called. That is, the calling procedure must know the element size of the return value (i.e., number of words comprising the value).

The following paragraphs describe an example of a procedure, named "f", that returns a value, and the instructions used to do so. The example is illustrated in Figure 4-32.

The procedure is of the form:

```
INT PROC f (x);
   INT x;

  BEGIN
    RETURN x * x;
  END;
```

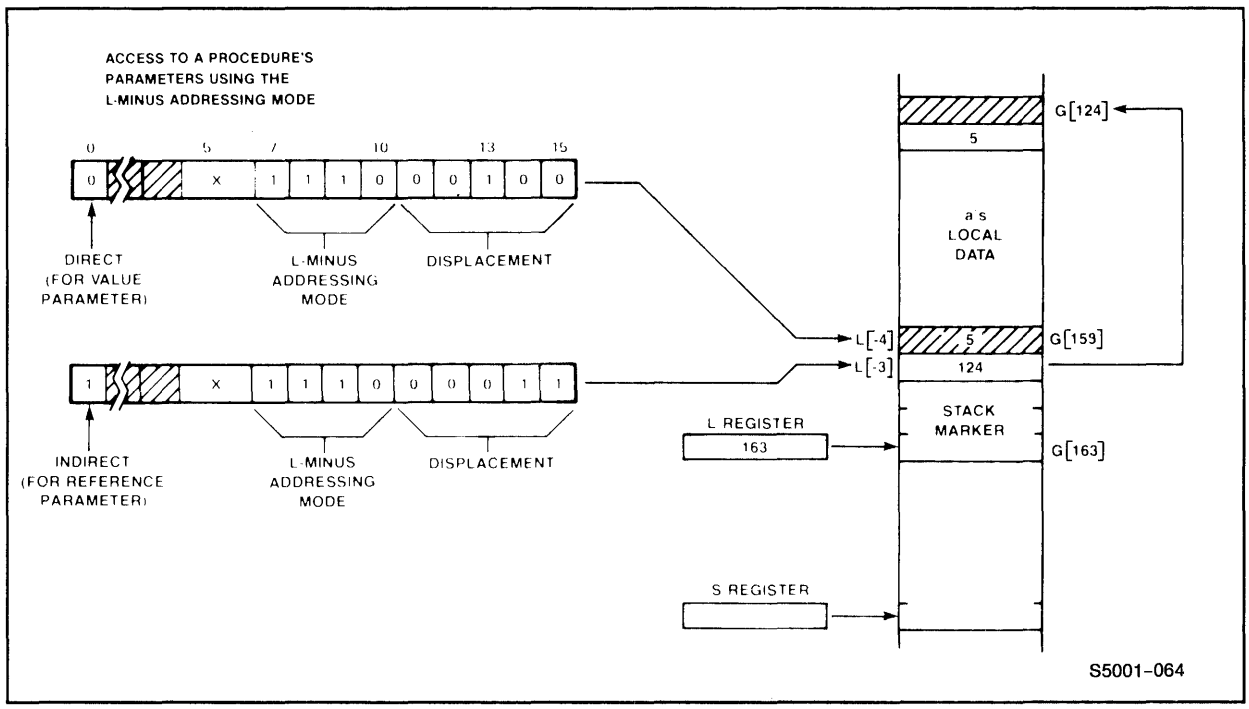


Figure 4-31. Parameter Access

This procedure returns the square of a number, "x". The instructions to return the square of "x" are:

```
LOAD L -003     !  parameter x is obtained from L-003
LOAD L -003     !  load another copy of x
IMPY            !  squared result now exists in R[0]
EXIT 4          !  delete stack marker and parameter x
```

The first LOAD instruction loads the parameter "x" onto the top of the Register Stack. Following the LOAD, the RP setting is 0. (The RP setting is 7 when a procedure begins executing.) The second LOAD again loads the parameter "x". Following this load, the RP setting is 1.

The IMPY instruction multiplies the values in the Register Stack, leaving the result of the multiplication in R[0]. Following this operation, the RP setting is 0.

The EXIT instruction causes a return to the caller, deleting the parameter and stack marker (1 + 3 = 4) from the data stack. The squared value is left on the top of the register stack.

Suppose a call is now made to procedure "f", as follows:

```
z := i + j - f(5);
```

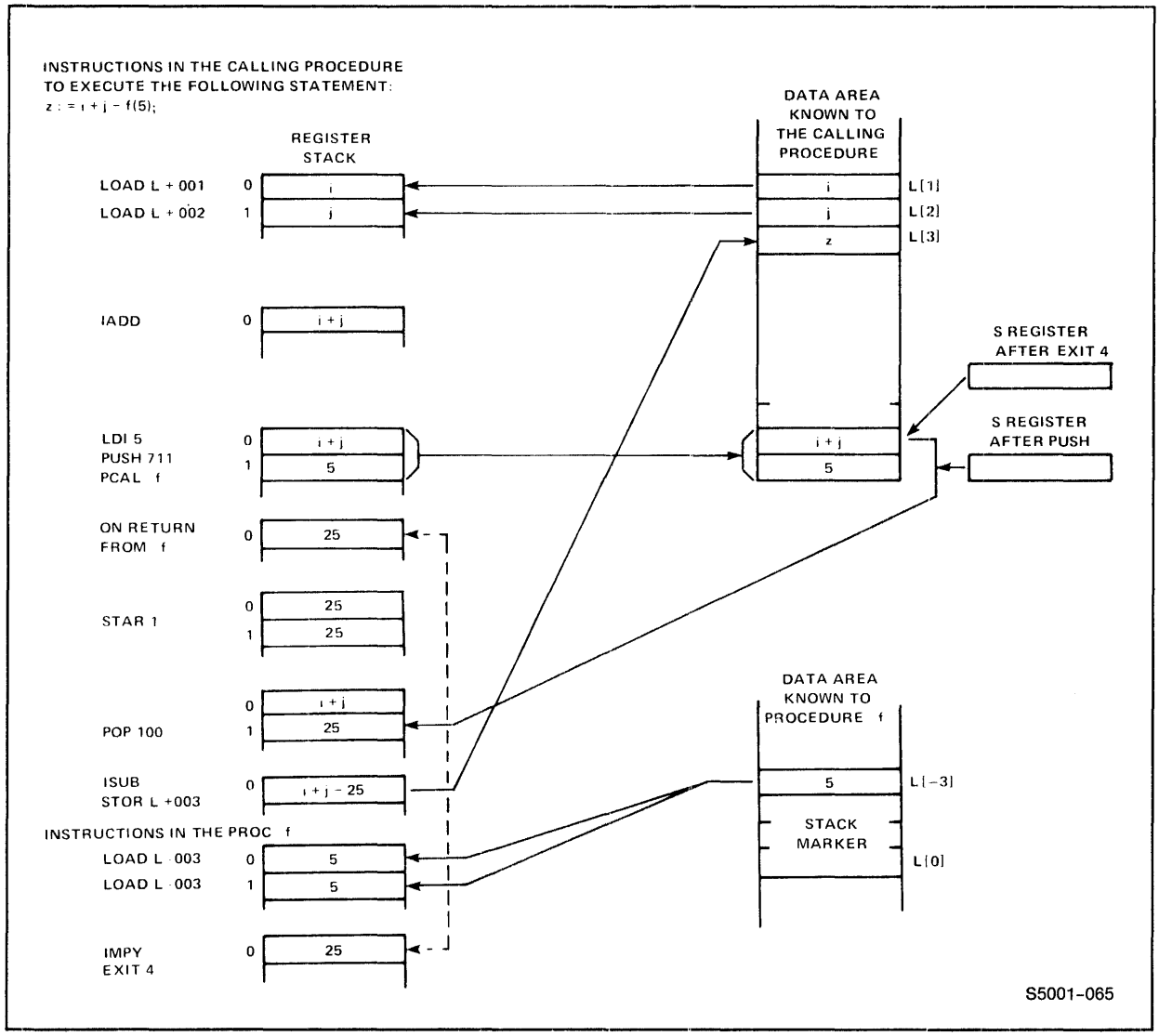


Figure 4-32. Value Returned Through Register Stack

That is, subtract the square of 5 from the sum of the contents of the variables "i" and "j" then store the result in the variable "z". Variables "i", "j", and "z" are local variables at L[1], L[2], and L[3], respectively.

The instructions to perform this operation are:

```
LOAD  L +001    ! load i
LOAD  L +002    ! load j
IADD            ! i + j
LDI     +005    ! load parameter to f
PUSH     711    ! push sum and parameter onto memory stack
PCAL            ! procedure call to f
STAR       1    ! move returned value from R[0] to R[1]
POP      100    ! bring saved sum back to R[0]
ISUB            ! subtract returned value from i+j sum
STOR  L +003    ! store result into z
```

The first three instructions calculate the sum of i + j and leave the result in R[0]. The LDI +005 instruction loads the parameter to "f" onto the top of the Register Stack at R[1].

The PUSH instruction pushes R[0:1] onto the memory stack. Following the PUSH, the two top-of-memory-stack locations contain:

```
S[-1] = sum of i + j
S[0]  = 5, the parameter to f
```

This clears the Register Stack for use by the procedure which now is invoked by the PCAL instruction. On the return from f, R[0] of the Register Stack contains the square of 5.

The STAR instruction moves the return value in the R[0] Register Stack location to R[1] in preparation for the subtraction from the sum of "i" + "j".

The POP 100 instruction brings the sum of "i" + "j" (calculated previously) into R[0] and sets RP to 1 (to point to the returned value).

The ISUB instruction subtracts the return value of "f" from the sum of "i" + "j". The STOR instruction stores the result in the variable "z", and RP becomes 7.

Stack Marker Chain

In examples shown previously, only one procedure call occurred, and, therefore, only one stack marker was generated. However, in practice, there can be several stack markers (and local areas) present in a memory stack at once. This occurs when a called procedure calls another procedure and that procedure calls still another procedure, etc. The nature of this "chain" of stack markers and the action of the L and S registers is such that the returns are always made in the reverse order of the calls, and the local data areas are redefined as the returns are made.

Figure 4-33 shows the condition of a memory stack after the following calls have taken place:

```
In procedure "a", CALL b;

    In procedure "b", CALL c;

        In procedure "c", CALL d;
```

The procedure "d" is currently executing.

Specifically, the L register, which is given a new (higher) setting when a procedure is called, and the local data areas, which are allocated and generated relative to the current L register setting, result in a stack of procedure environments that are physically placed in the chronological order in which the calls were made. (Remember, when a procedure is called, the stack marker is placed at the current S register setting plus one. In this manner, a procedure's local data is always retained when it calls another procedure.) The stack markers, which contain the environment of the preceding procedure (and point to the preceding stack marker) restore the preceding environments in the reverse order of the calls.

Subprocedures

Subprocedures are invoked using the BSUB (branch to subprocedure) instruction. Because the BSUB is a branching-type instruction, the subprocedure entry point is calculated as a self-relative address. Execution of the BSUB instruction differs from other branching instructions in that it places a return address on the top of the memory stack. See Figure 4-34. Note that before the BSUB executes, the subprocedure parameters must be pushed onto the stack.

Specifically, the steps involved when a BSUB instruction is executed are as follows:

1. The return address (i.e., that of the instruction following the BSUB) is placed on the top of the memory stack.

```
S := S + 1;
data[S] := P;
```

2. The self-relative branch address of the subprocedure is put into the P register.

```
P := branch^address;
```

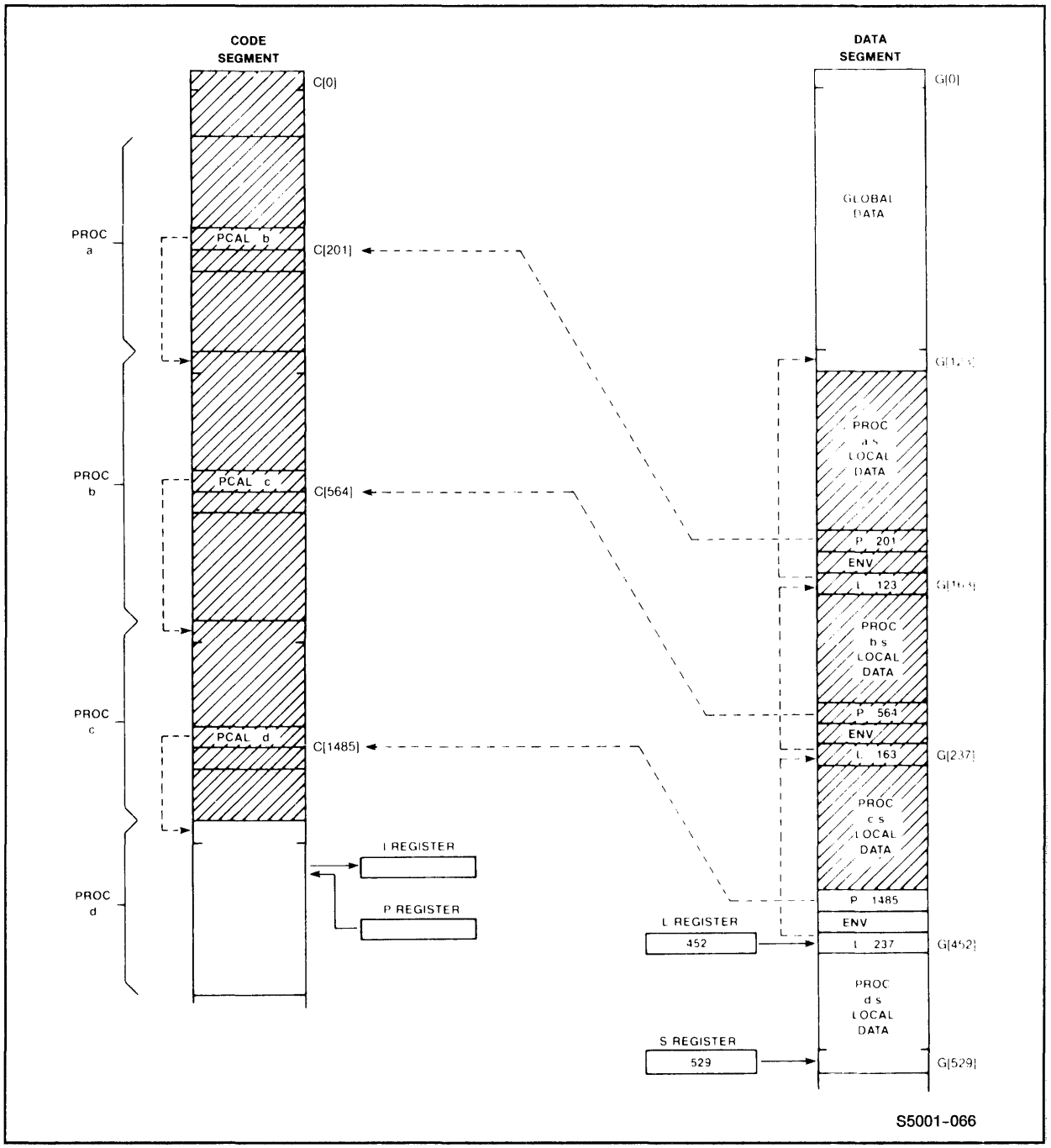


Figure 4-33. Stack Marker Chain

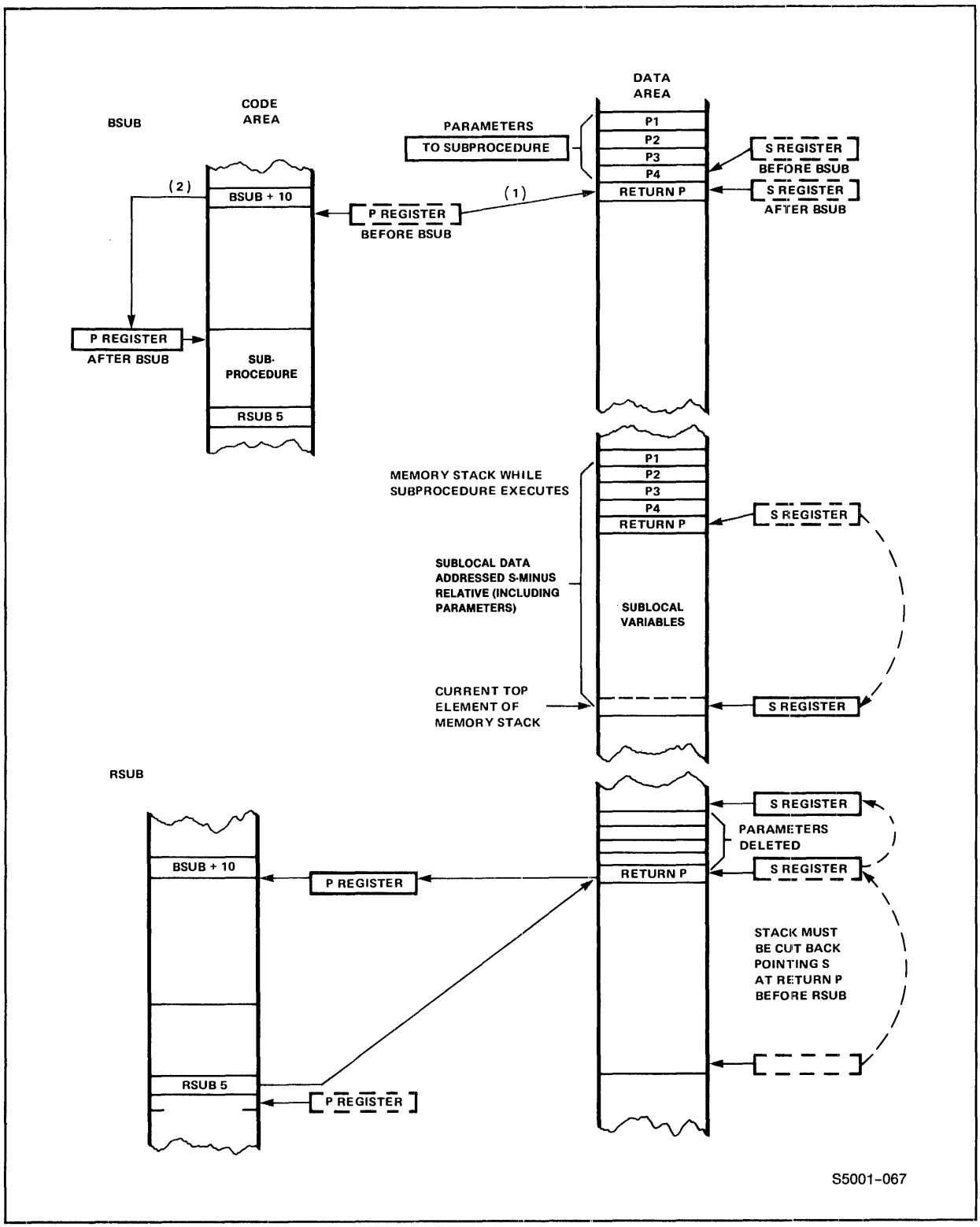


Figure 4-34. Subprocedure Calls

The last instruction that a subprocedure executes is an RSUB (return from subprocedure) instruction. The RSUB instruction returns control to the instruction following the BSUB instruction by putting the return address, at the current top of memory stack location, into the P register:

```
P := data [S];
S := S - S^decrement;
```

The "S^decrement" value (which is specified in the RSUB) is used to move the S register setting below the sublocal data area. "S^decrement" is at least one, to account for the one-word return address.

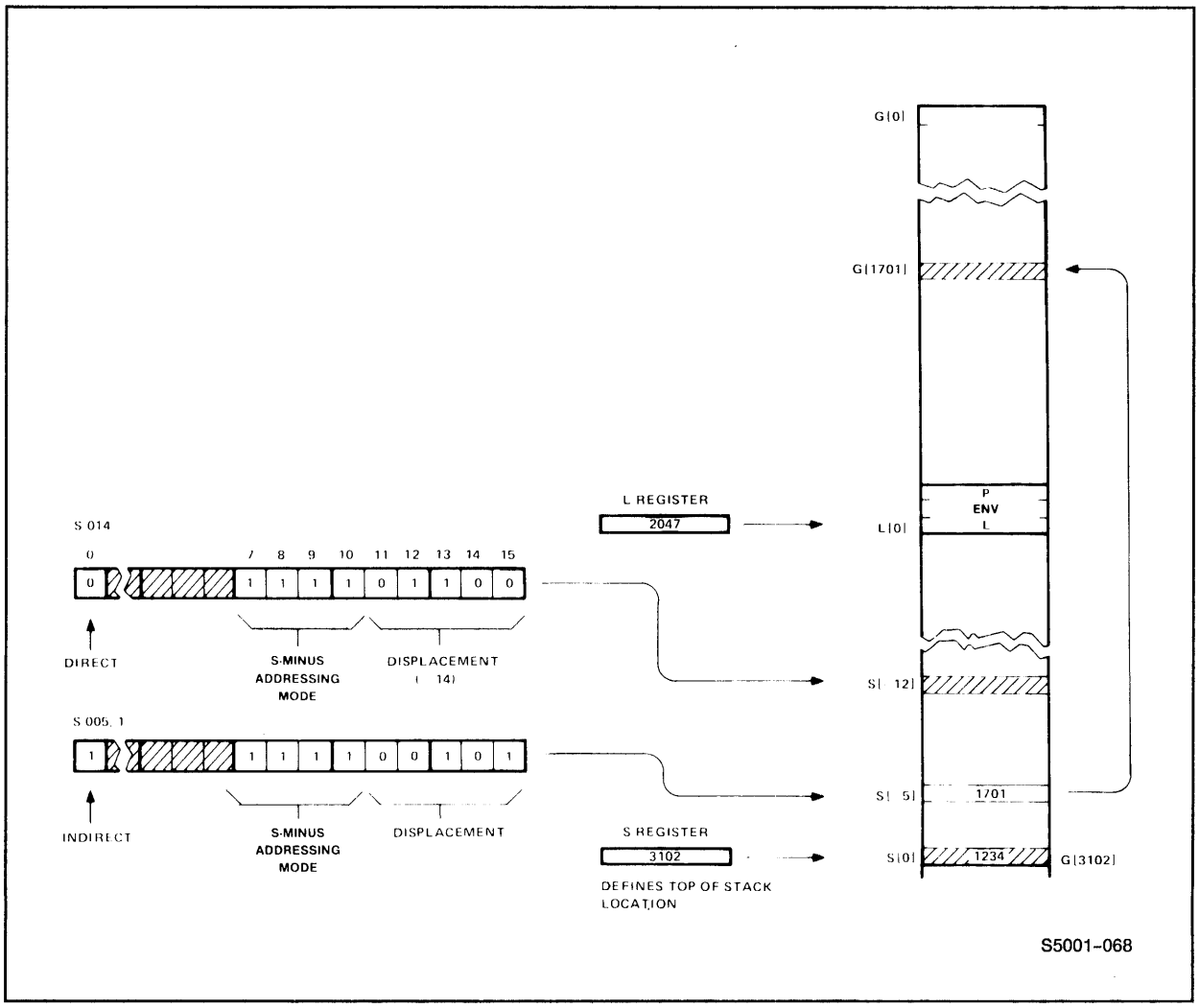


Figure 4-35. Example of S-Minus Addressing

The sublocal data area consists of a subprocedure's variables and parameters. It is addressable using the S-minus addressing mode, shown in Figure 4-35. This provides direct access to the 32 locations including and below the current S register setting (i.e., S[-31:0]).

SYSTEM GLOBAL ADDRESSING

If a system procedure must access the system data segment from the user environment, it is given the attribute "callable" (so that it can be called by the nonprivileged application program) and executes in privileged mode. Executing in privileged mode permits the procedure to make use of the SG-relative (system global relative) addressing mode. This addressing mode, illustrated in Figure 4-36, provides access to the system data

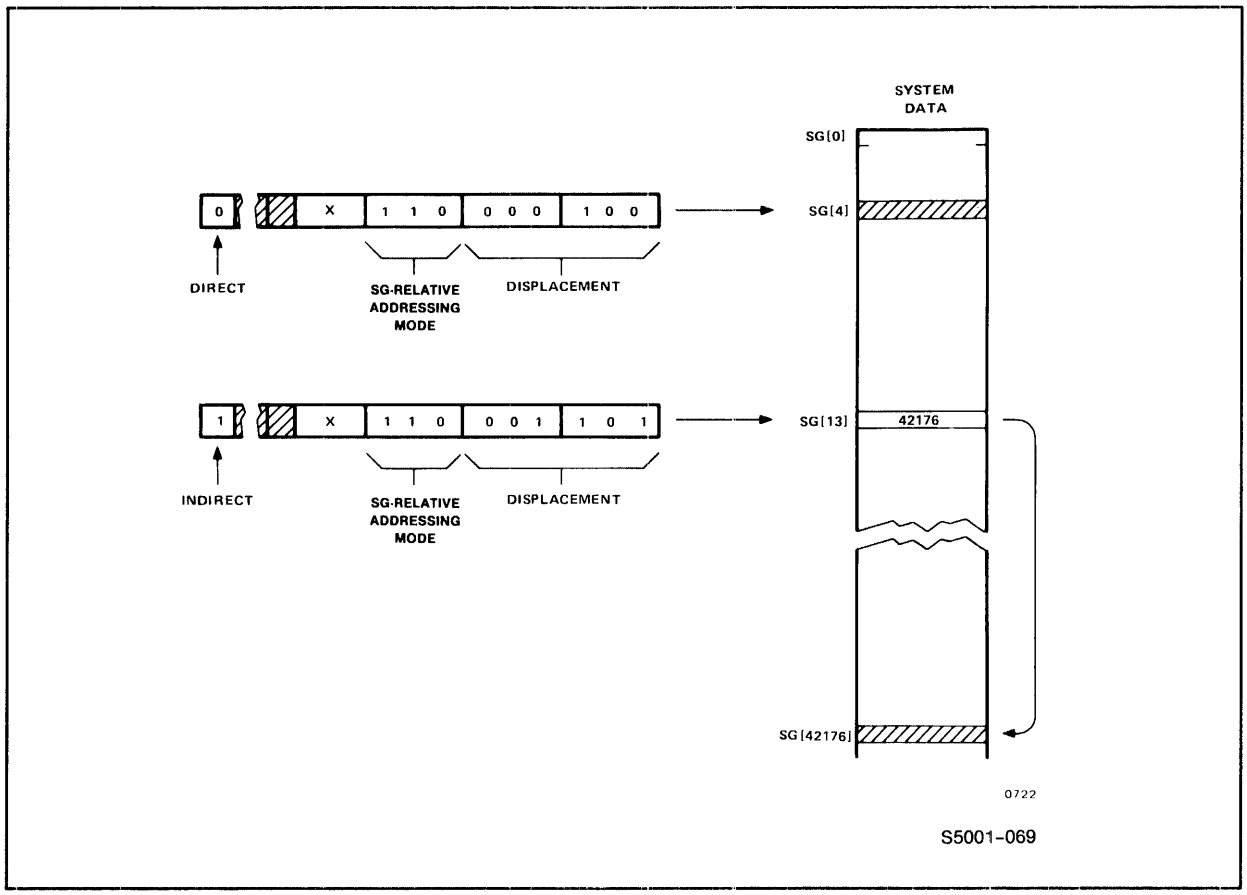


Figure 4-36. SG-Relative Addressing Mode

segment (and, therefore any system tables) even when DS indicates user data.

The SG-relative mode for a memory reference instruction allows direct addressing of the first 64 locations of the operating system's data segment (SG[0:63]). This mode is indicated when bits I.<7:9> of the memory reference instruction are equal to 110. Bits I.<10:15> are a positive word displacement from SG[0]:

```
direct^address = I.<10:15>
```

The short address space used for the SG-relative addressing mode is determined by the function:

```
short address space:
  if I.<7:9> = 6 and PRIV then 1      ! system data
                          else DS;    ! current data
```

Indirect addressing and indexing are both permitted with the SG-relative addressing mode. Executing in privileged mode while in the user environment also means that data can be moved, compared, and scanned (with the MOVW, MOVB, COMW, COMB, SBW, and SBU instructions) between the user data segment and the system data segment.

SECTION 5

ADDRESSING AND MEMORY ACCESS

This section discusses the form of physical and logical addresses and the relationship between these address types. Mapping--that is, how memory is actually accessed using these addresses--is also described. This level of information is useful to most systems programmers and some applications programmers.

NOTE

Throughout this discussion, the suffix "k" represents the number 1024, and the prefix "mega", when applied to a number of bytes or words, represents 1k squared, or 1,048,576. Likewise, the prefix "giga" means 1k cubed, or 1,073,741,824.

PHYSICAL, VIRTUAL, AND LOGICAL MEMORY

Physical memory is the semiconductor memory storage that is provided by each processor's own memory boards (one to four). A processor module's physical memory address space can consist of up to 8,388,608 words (8 megawords) of 16 bits each. (The maximum physical memory presently available is 4 megawords.) Physical memory is divided into contiguous blocks called "physical pages." (A page is a block of 1024 consecutive words, or 2048 consecutive bytes.) The maximum physical memory address space is, therefore, 8192 pages.

Pages in physical memory are numbered consecutively from page 0 to page 8191. Words in physical memory are numbered 0 through 8,388,607 and are addressed with a 23-bit physical address. The range of physical memory is shown in Figure 5-1; the format of a 23-bit physical address is illustrated in Figure 5-2. Note that,

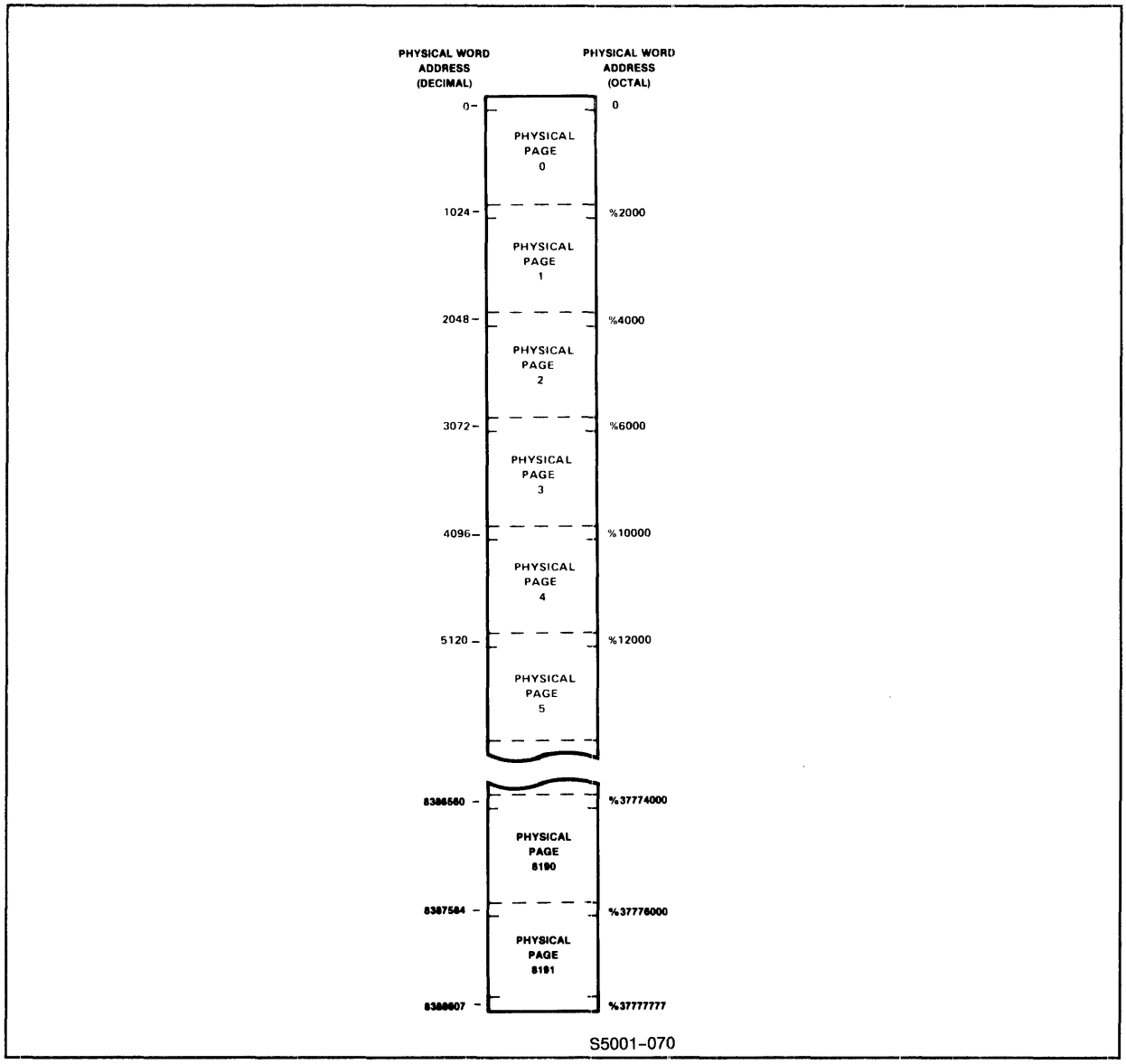


Figure 5-1. Physical Memory

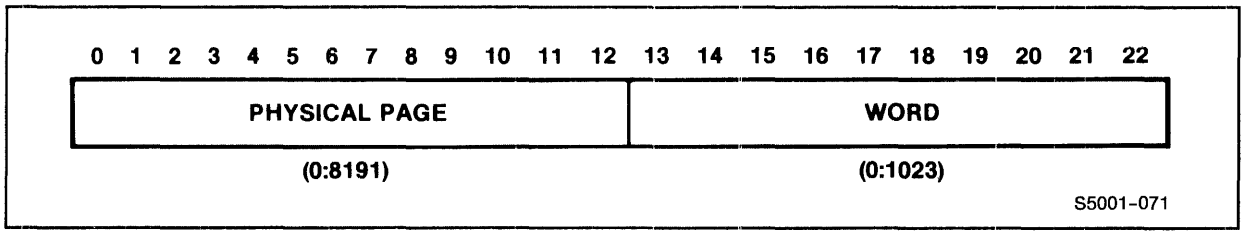


Figure 5-2. 23-Bit Physical Address

as shown in Figure 5-2, bits 0 through 12 of a physical address specify the number of the physical page, and bits 13 through 22 specify the word within the page.

Note that the physical address does not specify a byte. Memory references are controlled by the Memory Control Unit (MCU), which cannot perform byte addressing; it can address only full words on word boundaries. Addressing of a particular byte within memory is done by the IPU microcode.

In the NonStop II processor, physical memory is accessed by logical pages mapped either in maps 0-14 or in the extended address cache. The maps translate logical page numbers to physical page numbers. There is no caching of data or instructions.

In the NonStop TXP processor, physical memory is accessed by logical pages mapped in a page table entry cache (PCACHE). PCACHE translates logical page numbers to physical page numbers. If the desired information is already present in the data cache (CACHE), it is not necessary to go through the process of translating the logical address to a physical address and accessing physical memory.

Virtual memory utilizes disc space to extend the storage space that is accessible in physical memory. In a multiprogramming environment, the total memory space needed for all processes and the operating system usually exceeds the physical memory available. However, at any moment, only a subset of the total is required for continued operation. Images of memory pages are maintained in disc storage and are brought into physical memory as required by process execution. These disc images can be either code or data. Data images not currently required for execution of a given program can be "swapped out" (returned to disc) so that their physical memory can be used by another process. Because code images cannot be modified, it is not necessary to return current copies of them to disc before giving their physical memory space away.

The virtual memory for a given processor is the sum total of all code and data images that can possibly be brought into its main memory. To provide addressability to this entire range of virtual memory, a processor's virtual memory is divided into 8192 blocks called segments (or, more specifically, "absolute segments"). See Figure 5-3. Each segment can be up to 64 pages in length, or 64k words.

Individual segments may be unallocated (not presently in use) or allocated. A segment that is allocated can have fewer than 64 pages in use. In such a case, the entire segment address space is reserved (that is, no address within that segment can be used

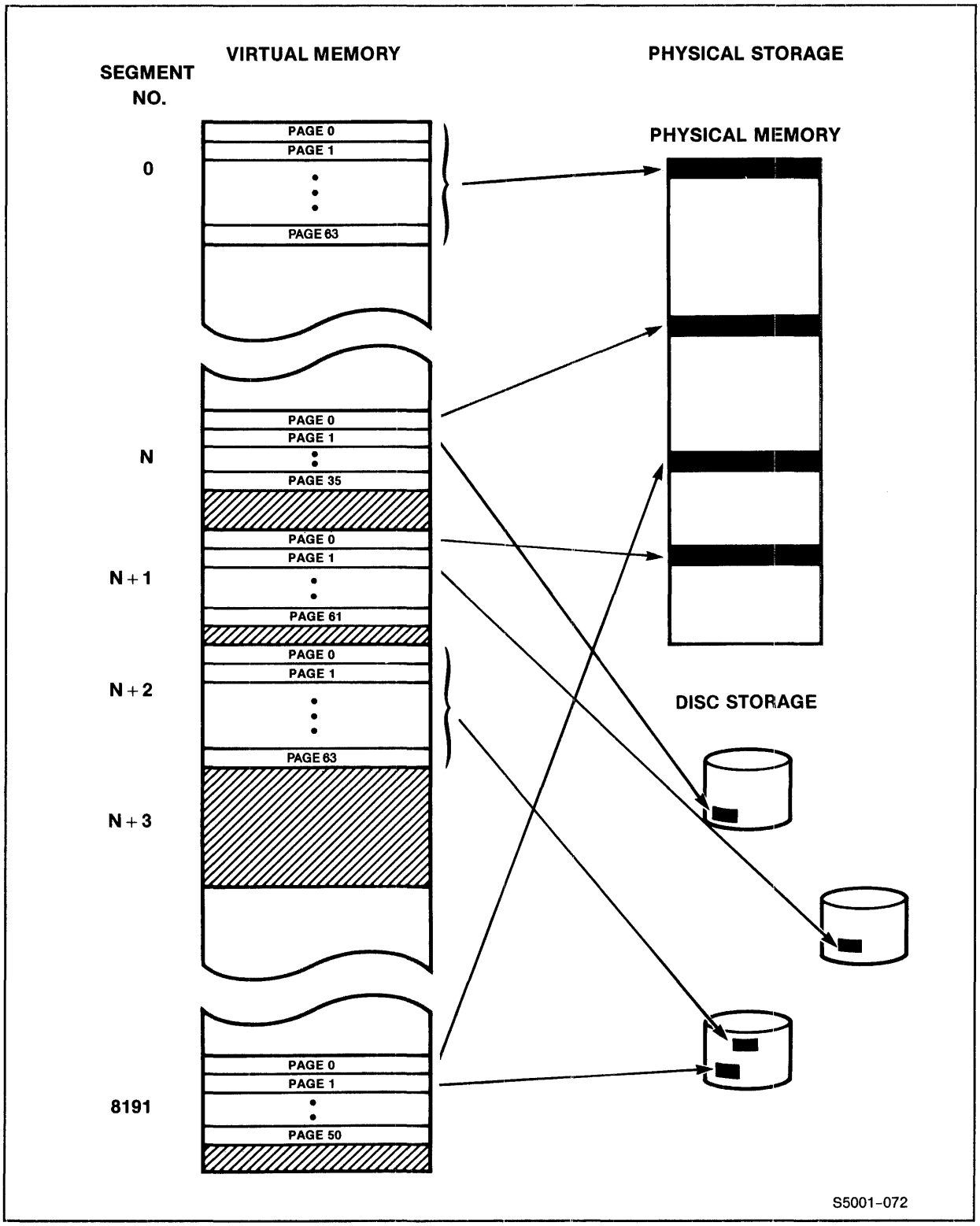


Figure 5-3. Virtual Memory

by any other process), but the process that owns the segment can use only as many memory pages as it requested. Also, only individual pages of a segment are brought into main memory, rather than the entire segment--and only as needed. This also is illustrated in Figure 5-3.

A privileged process can see and access all of virtual memory using 32-bit absolute addresses. Absolute addresses are described later under "Extended Addressing".

Logical memory is memory as a process views it. In general, a process is allowed to see only a subset of virtual memory, consisting of code and data areas that it owns or shares. It usually does not matter to a process whether the words being addressed are present in physical memory or are absent (stored on disc). The process simply uses logical addresses that are valid within its own set of addressable segments. The operating system takes care of bringing in absent pages as needed.

For nonprivileged processes, logical memory is separated into six "short address" spaces (addressable with either 16-bit logical addresses or 32-bit relative addresses) and one "extended address" space (addressable only with 32-bit relative addresses). The extended address space is considered later under "Extended Addressing." The six short address spaces (SASs) accessible to a process consist of the following (refer to Figure 5-4):

Space	Description	
0	User Data	(one segment per process)
1	System Data	(one segment per CPU)
2	User Code	(1 to 16 segments per process)
3	System Code	(one segment per CPU)
4	User Library	(0 to 16 segments per process)
5	System Library	(1 to 32 segments per CPU)

The odd-numbered short address spaces (1, 3, 5) belong exclusively to the GUARDIAN operating system. SAS 1 is always the system data segment; this segment contains various system values and tables, and is accessible by all processes. (Such access is usually performed by the system on behalf of the process, since the DS or PRIV bit in the Environment Register must be set in order for the access to be allowed.) SAS 3 and SAS 5, system code and system library, contain the system procedures and interrupt handlers (but not the program code for system processes) of the GUARDIAN operating system. Many of these procedures are callable by any process; others require privileged mode. As indicated in Figure 5-4, there can be up to 33 segments for these procedures and interrupt handlers: one system code segment, and up to 32 system library segments.

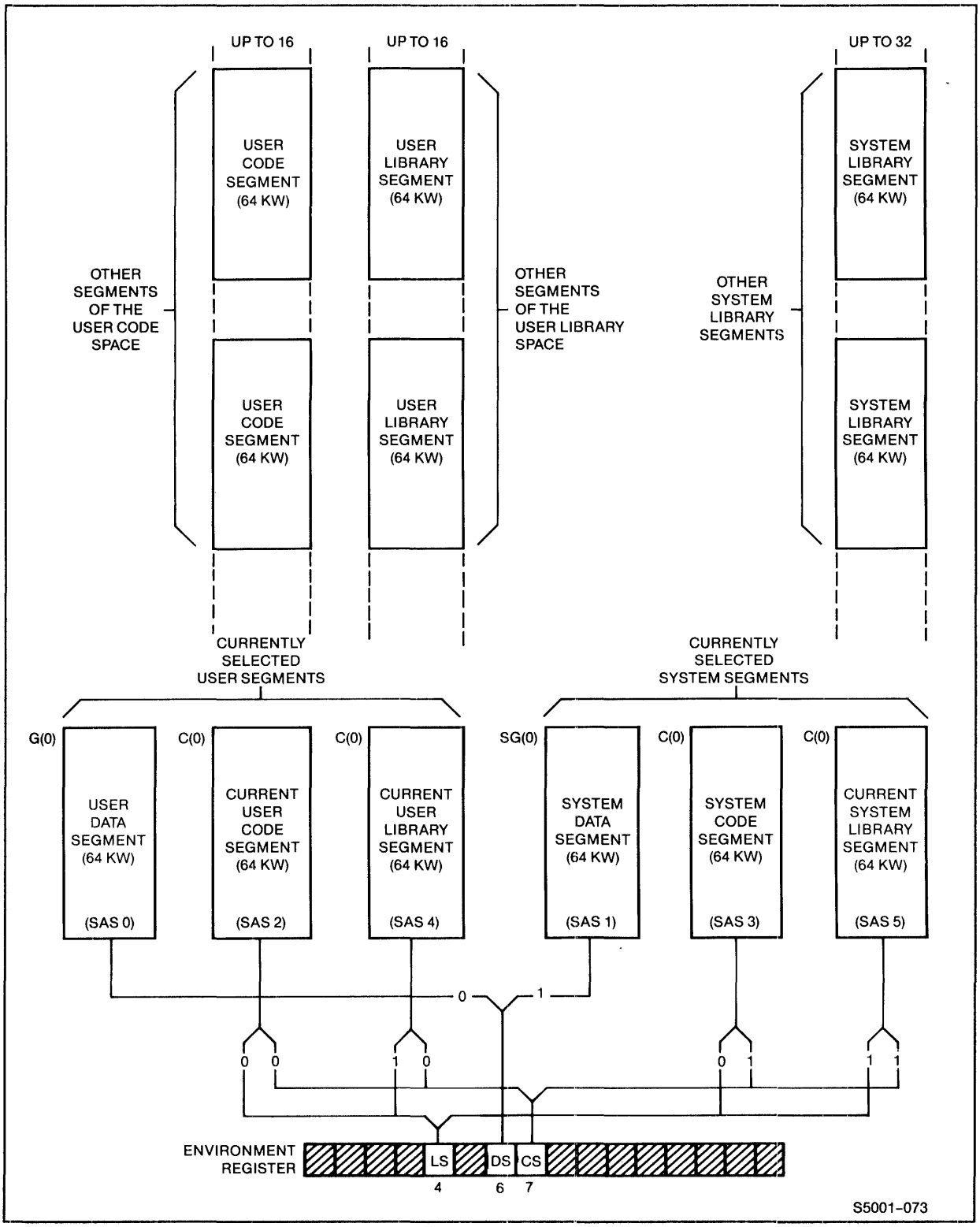


Figure 5-4. Logical Memory

Generalized terms such as "the system library" or "the system code space" refer to the entire set of up to 33 segments in SASs 3 and 5.

NOTE

Segment selection for the multiple-segment code spaces is performed by the microcode of the XCAL, DPCL, EXIT, and IXIT instructions, using a space ID index value (as described earlier in Section 4, under "Procedures and the Memory Stack"). The currently-selected segment for each space is identified in a table in system data space (CSSEG, Current Short-address Segments). The NonStop TXP processor also retains this information in a set of hardware registers called the Short Segment Table (SST).

The even-numbered short address spaces (0, 2, 4) contain the code and data of the currently executing process. Since many processes typically exist in a processor (such as user application processes, I/O processes, compiler processes, and GUARDIAN processes), the actual code and data indicated by these spaces switches each time a different process comes into execution. Every such process performs its addressing relative to its own G[0] and C[0] bases. As indicated in Figure 5-4, the user code space can consist of up to 16 code segments (that is, 2 megabytes), and the user library space provides an additional 16 segments for library procedures.

Any single memory-reference instruction can access only one code segment and one data segment. Their selection, from among the six short address spaces in logical memory, is made by the existing state of three bits in the Environment Register; in the case of multisegment code spaces, further resolution is made by the space ID index (discussed earlier) to select one segment within the short address space. As shown in Figure 5-4, the selection of a data segment is made by the state of the DS bit (bit 6). If DS is equal to 1, the system data segment is accessed by the instruction; if DS is equal to 0, the user data segment is accessed. The selection of a code space is made by the combined settings of the LS and CS bits, as follows:

LS	CS		
0	0	User Code	(SAS 2)
0	1	System Code	(SAS 3)
1	0	User Library	(SAS 4)
1	1	System Library	(SAS 5)

16-BIT ADDRESSING

The memory area addressable by 16-bit addresses is limited, being applicable only in the six short address spaces. Since this mode of access is fast and efficient, the six address spaces most important to the execution of a process are made accessible with this type of addressing. 16-bit addresses are actually a kind of shorthand, and additional information is needed to identify the address space. The addressing modes described earlier under "Program Environment" (G-relative, L-plus-relative, L-minus-relative, S-minus-relative, and SG-relative) all use 16-bit addresses.

The IPU hardware uses the currently executing instruction and the LS, DS, and CS bits of the ENV register to select one of the six short address spaces. These address spaces are: user data, system data, user code, system code, user library, and system library. Since the SAS numbers for these address spaces are only in the range of 0:5, only three bits are needed to identify the space number.

The range of addressing within any of these six spaces is that of the 16-bit logical address, 0 through 65,535. Access to these six spaces is described later under the headings "Memory Access (NonStop II Processor)" and "Memory Access (NonStop TXP Processor)".

The formats for 16-bit addresses are shown in Figure 5-5.

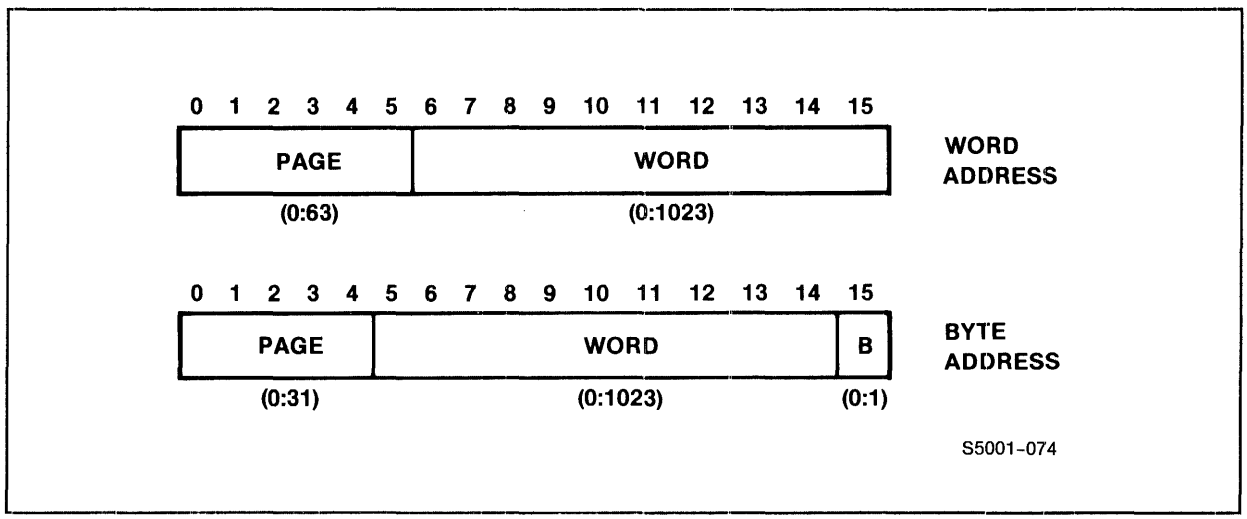


Figure 5-5. 16-Bit Logical Address

As shown, a 16-bit address can take one of two forms, depending on whether the instruction being executed is a word-addressing instruction or a byte-addressing instruction. For word access, the first six bits (0 through 5) specify the logical page number (0 to 63). Bits 6 through 15 then specify which of the 1024 words on that page is desired. For byte access, bit 15 is used to specify a particular byte within the word: 0 for the left byte or 1 for the right byte. The other fields appear one bit to the left of their positions in the word address, making the page field one bit smaller. Thus only the first 32 pages of a data segment--that is, the first 32768 words of the segment--can be accessed by byte. (For code addressing, however, either half of the segment can be accessed, since the address is taken to be in the same 32-page half of the segment as the current setting of the P Register.)

EXTENDED ADDRESSING

Extended addresses provide a uniform method of addressing all items in virtual memory. An extended address is 32 bits long; its format is shown in Figure 5-6.

Bit 0 (the absolute bit) indicates whether the address is an absolute extended address (bit 0 = 1), or a relative extended address (bit 0 = 0). Bit 1 is reserved and must always be zero. Bits 2 through 14 (the segment field) define which of the 8192 possible segments is being addressed.

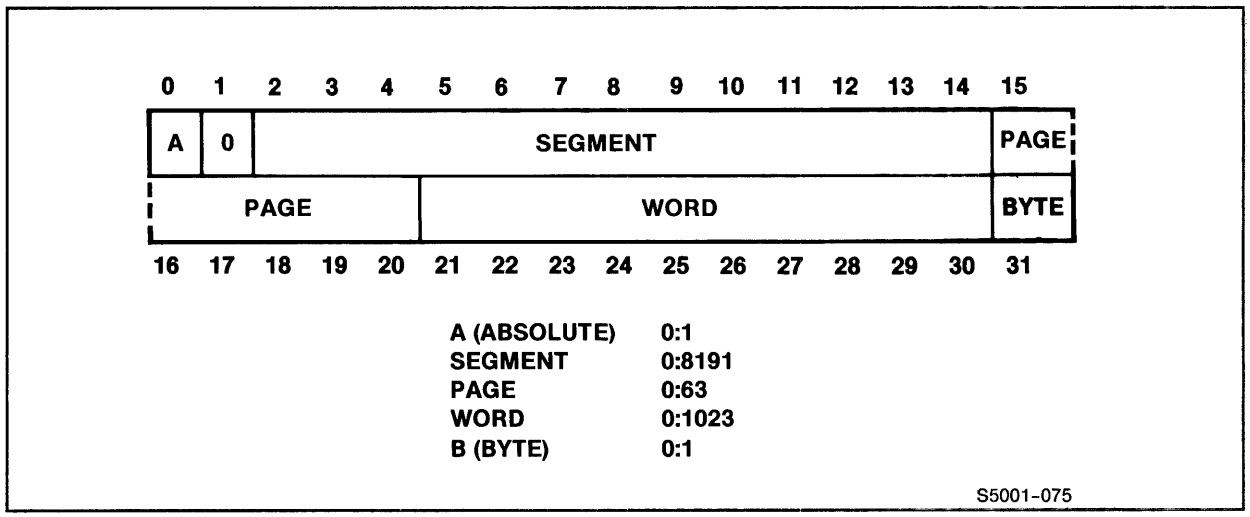


Figure 5-6. 32-Bit Extended Address

Bits 15 through 20 (page field) define which of the 64 pages in the segment is selected. Bits 21 through 30 (word field) specify which of the 1024 words in the page is addressed. Bit 31 (byte field) defines which byte of the word (0 for left or 1 for right) is to be accessed. Extended addresses are always byte addresses.

Absolute extended addresses can be used only by privileged processes. If a process executing in nonprivileged mode attempts to use an absolute extended address, an instruction failure occurs. In an absolute extended address, the segment field represents the number, 0 to 8191, of an absolute segment.

Relative extended addresses can be used in any program, privileged or nonprivileged. A relocation mechanism is provided for these addresses, so that all processes can use the same range of addresses, relative to a base address of 0. To use a relative extended address, the processor must translate it into an absolute extended address. In a relative extended address, the segment field represents the number of a relative segment. Each accessible relative segment is mapped onto one absolute segment.

A process's currently accessible short address spaces--namely, current data, system data, current code, user code--can be accessed with 32-bit relative addresses as relative segments 0 through 3, respectively. The selection of "current data" and "current code" for relative segments 0 and 2 are determined by the current bit settings of the ENV register, as illustrated earlier in Figure 5-4. System data is always accessible as relative segment 1. Relative segment 3 provides access to the currently mapped user code segment; this is useful for accessing code segment arrays, analogous to the "UC" mode addressing with 16-bit addressing (see footnote of Table A-5 in Appendix A).

Figure 5-7 illustrates relative extended addressing in segments 0 through 3. Note that with relative extended addresses, all (or as much as exists) of a data segment can be byte-addressed--not just the first 32k words as with 16-bit addresses.

The more significant application of 32-bit relative addresses, however (besides addressing the short address spaces), is to provide access to the extended address space. This space, for each process, consists of the relative segments numbered 4 and up (to a maximum segment number of 1027).

The extended address space is used (by both NonStop II and NonStop TXP processors) to contain an extended data segment. A process may allocate one or more extended data segments to contain large blocks of process data (up to 128 megabytes each). Although the process may have several extended data segments, only one at a time may be in use. Only the current one is effectively included in the process's logical memory.

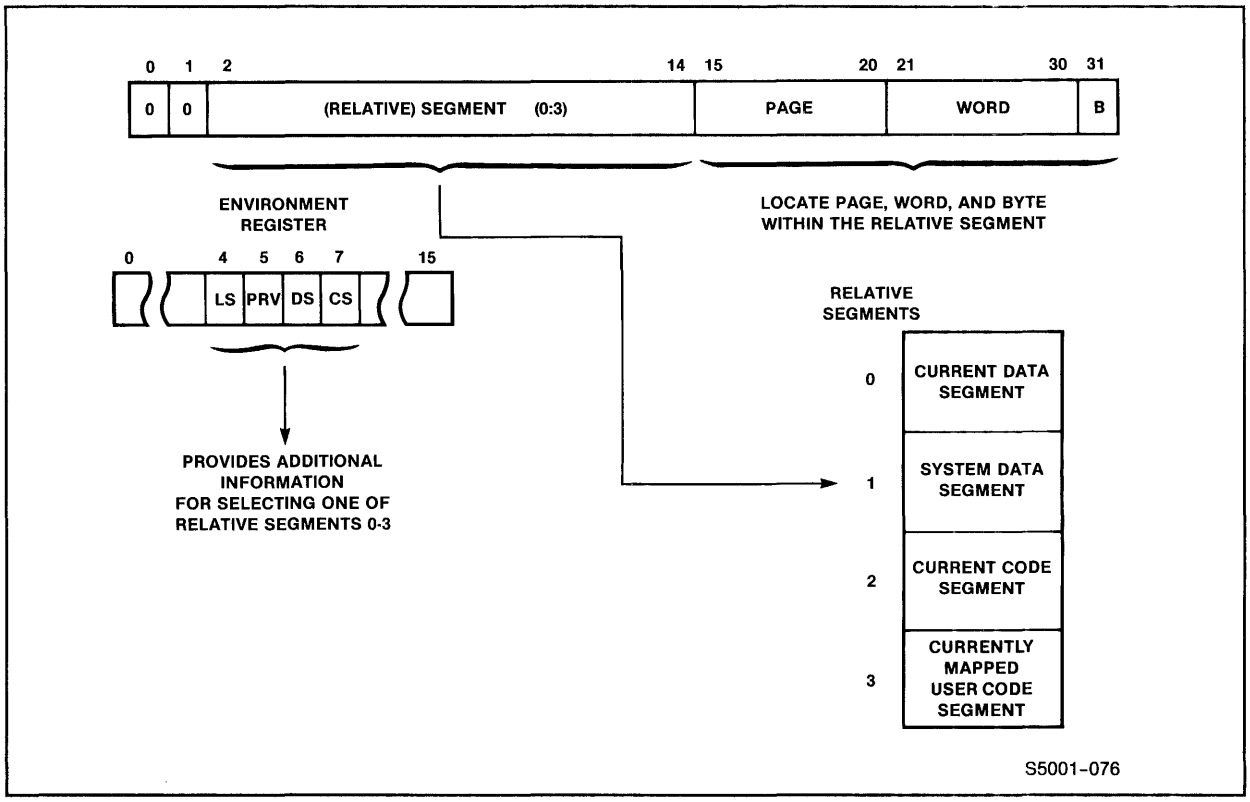


Figure 5-7. Relative Extended Addressing in Segments 0 through 3

NOTE

In this manual, the word segment generally means a nonextended segment (that is, a simple 1K to 64K word segment) except where the word "extended" is specifically used.

The base address of an extended data segment is always relative segment 4, page 0, word 0, byte 0. The relative addresses within an extended data segment are consecutive, no matter how large it is; for example, segment 5, page 1, word 0, byte 0 refers to the first byte of the 66th page of the extended segment. A relative extended address with a segment number of 4 or greater always refers to a location within an extended data segment.

An extended data segment is always allocated as a contiguous block of absolute segments, so that a simple relocation mechanism can be used. See the discussion of base and limit that follows.

To request allocation of an extended data segment, a process calls the operating system procedure ALLOCATESEGMENT. Once the segment has been allocated successfully, it must be put in use by a call to the USESEGMENT procedure before it can be accessed. A process can have several extended data segments, but only one can be accessed at any given time; a new call to USESEGMENT must be made each time a different extended data segment is to be used. A call to DEALLOCATESEGMENT (or stopping the process) frees a segment when it is no longer needed.

Figure 5-8 illustrates three extended data segments belonging to a process.

In extended data segments, two special values--the segment base and limit--are used to determine the absolute virtual memory location represented by a relative extended address. The base defines the beginning of the relative segment; it is the absolute extended address of the first byte in the relative segment, minus %2,000,000 (%2,000,000 is the address of the beginning of segment 4). The limit defines the maximum relative address that can be used within the segment. For efficiency, the limit is stored as

- (segment size in bytes + %2000000)

so that the following algorithm can be used in resolving a relative extended address: First, the limit is added to the address. If the result is large enough to cause a carry, the address is out of bounds, and an instruction failure trap occurs. Otherwise, the relative extended address and the base are added together to produce an absolute extended address. See Figure 5-9.

The base and limit for an extended data segment are determined when a process requests allocation of that extended segment. The limit is determined from the segment size specified by the user process; the base is determined by the operating system.

Extended Addressing Instructions

The NonStop II and NonStop TXP processors provide a class of instructions to access data using extended addresses. An example is the MVBX instruction, which allows bytes to be moved between any two locations in virtual memory.

The following is a list of extended addressing instructions. These 23 instructions are nonprivileged, and most (all except MNDX, XSMX, and CDX) are supported by TAL language constructs. For detailed descriptions of these instructions, refer to Section 9, "Instruction Set."

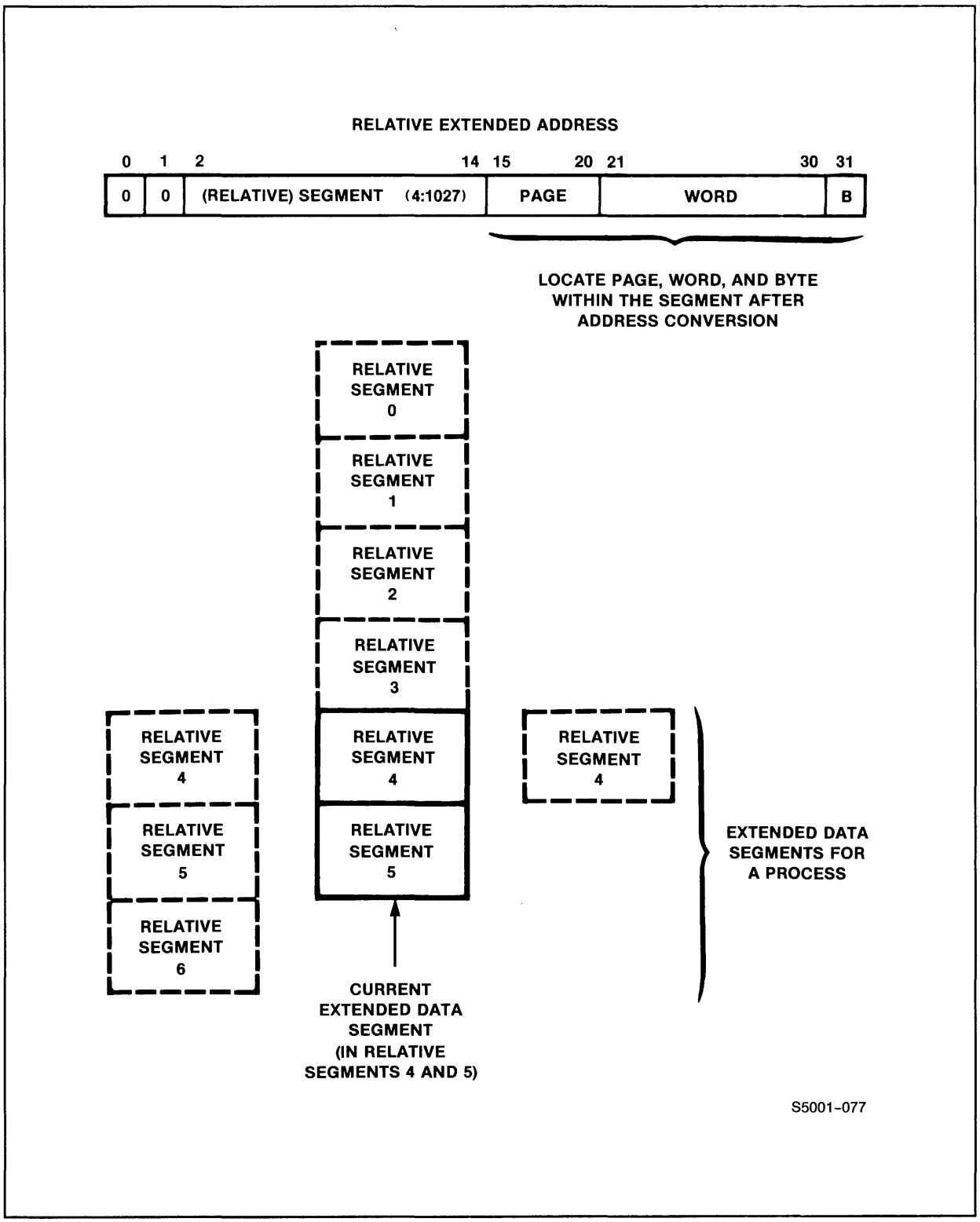


Figure 5-8. Relative Extended Addressing in Segments 4 and Up

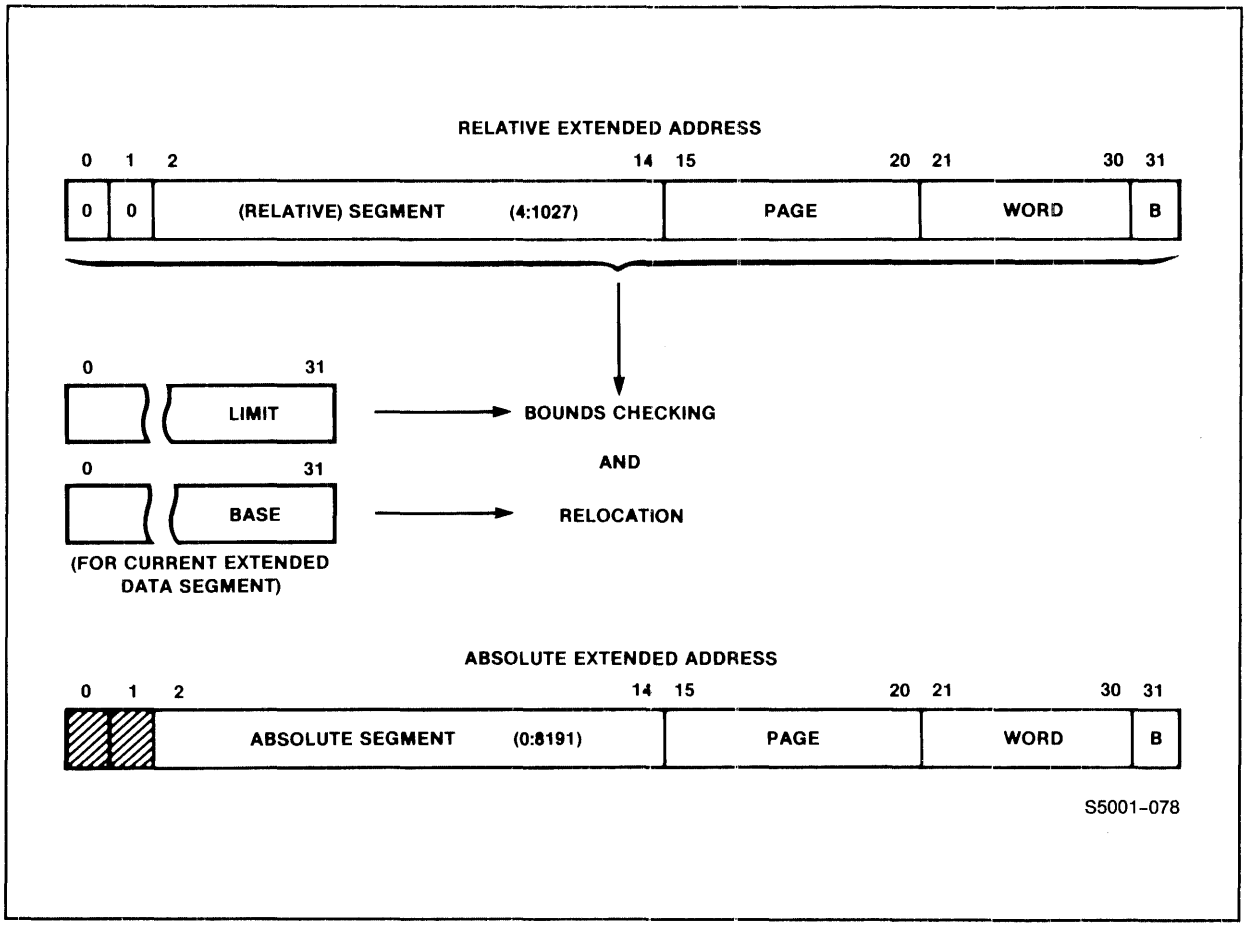


Figure 5-9. Address Conversion for Relative Segments 4 and Up

ANX	AND to Extended Memory
ORX	OR to Extended Memory
MNDX	Move Words While Not Duplicate
XSMX	Checksum Extended Block
CDX	Count Duplicate Words Extended
LBX	Load Byte Extended
SBX	Store Byte Extended
LWX	Load Word Extended
SWX	Store Word Extended
LDDX	Load Doubleword Extended
SDDX	Store Doubleword Extended
LQX	Load Quadrupleword Extended
SQX	Store Quadrupleword Extended
DFX	Deposit Field Extended
MVBX	Move Bytes Extended
MBXR	Move Bytes Extended, Reverse
MBXX	Move Bytes Extended, and Checksum

CMBX	Compare Bytes Extended
SCS	Set Code Segment
LWXX	Load Word Extended, Indexed
SWXX	Store Word Extended, Indexed
LBXX	Load Byte Extended, Indexed
SBXX	Store Byte Extended, Indexed

MEMORY ACCESS (NonStop II PROCESSOR)

This subsection describes the actual mechanisms used to access memory in the NonStop II processor. This information is primarily needed by systems analysts, though it may also be of interest to others. A parallel subsection dedicated to the NonStop TXP processor follows this description.

Maps

A processor module converts 16-bit logical addresses and 32-bit absolute extended addresses to 23-bit physical addresses by means of mapping, a method which uses a set of special map registers in the processor. Each processor in a NonStop II processor has sixteen maps, each map consisting of 64 map registers.

Maps 0 through 5 provide address translation for the six short address spaces that are accessible to the current process (illustrated earlier in Figure 5-4). Each of these six maps is capable of mapping a logical segment (up to 64 pages); each map register contains the starting address in physical memory of one page of the segment. The remaining ten maps define other segments (not accessible to most processes) or have other specialized purposes.

Figure 5-10 shows the uses of all sixteen maps and compares them to the uses of the first sixteen absolute segments. As shown, some maps correspond to the absolute segments of the same numbers. The uses of the maps are as follows:

0 User Data. This map defines the data segment of the currently executing process; that is, it maps the physical location of each page of the segment that is assigned to be the current process's data space. If the DS bit of the ENV Register is set to 0, all data references are directed into the segment defined by this map unless they are made by instructions which use either extended addresses or the SG-relative addressing mode.

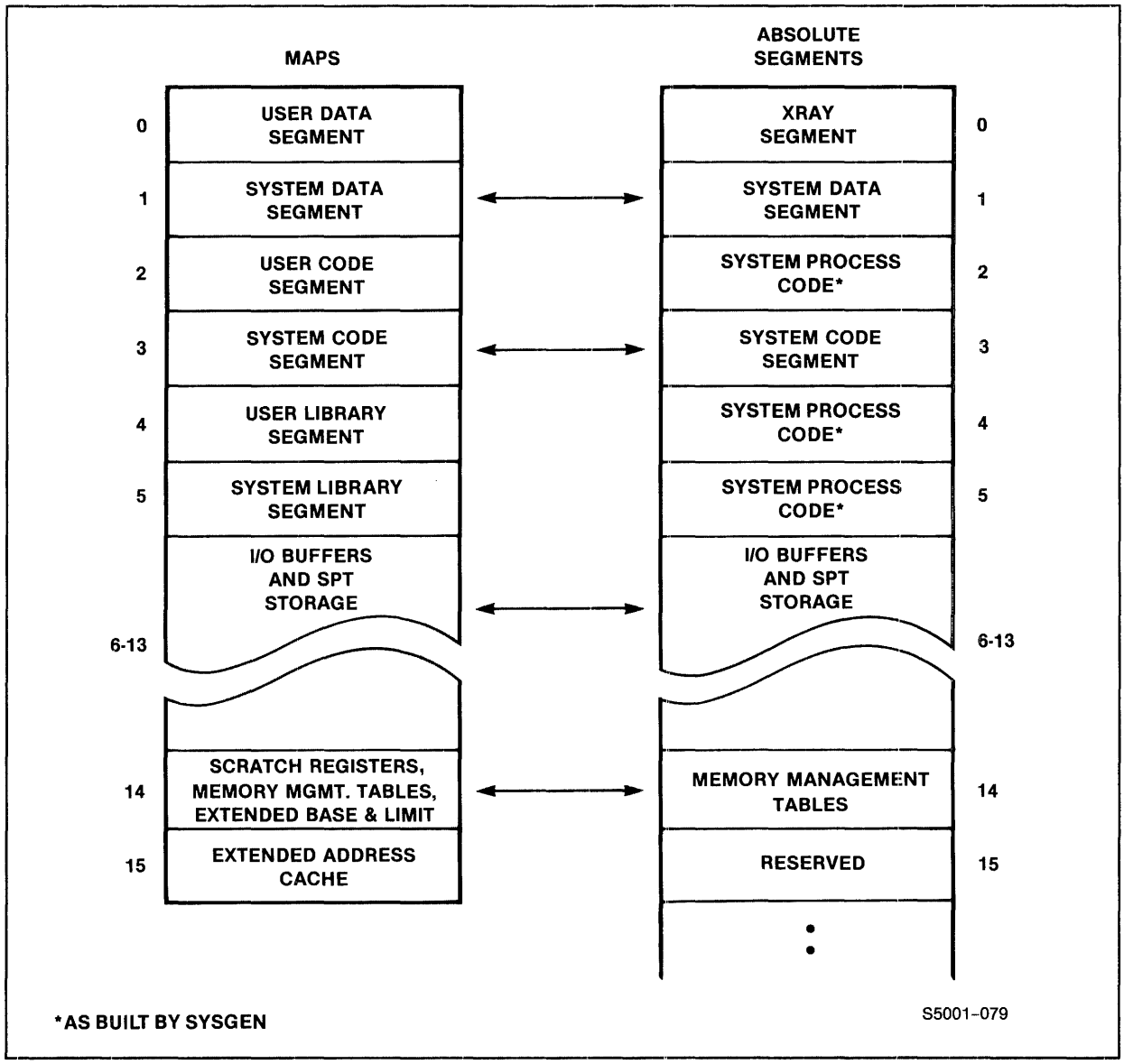


Figure 5-10. Uses of Maps and Absolute Segments

1 System Data. This map defines the segment that contains system tables and stacks for the interrupt handlers. The space defined by this map is common to all processes, but it may be accessed only if the DS bit in the ENV register is set to 1, or if explicit reference is made to the system data segment (for example, with SG-relative addressing) and the PRIV bit is set. This space is always absolute segment 1.

2 User Code. This map defines the current code segment of the currently executing process; that is, it maps the absolute segment invoked by the most recent call instruction (XCAL or DPCL) or return instruction (EXIT or IXIT) to or from these segments. All code references specify this segment if the CS and LS bits in the ENV register are 0. In addition, "UC" mode addresses always reference this segment regardless of the ENV register bit settings.

3 System Code. This map defines the code segment for operating system code (interrupt handlers and frequently used system procedures). The space defined by this map is common to all processes, and is always absolute segment 3. All code segment references (except "UC" mode addresses) specify this segment if the LS bit in the ENV register is 0 and the CS bit is 1.

4 User Library. This map defines the current user library segment for the currently executing process, if one exists for that process. It maps the absolute segment invoked by the most recent call instruction (XCAL or DPCL) or return instruction (EXIT or IXIT) to or from these segments. User library segments are mapped "on demand"; until there is a library call, none of these segments is current. All code references (except "UC" mode addresses) specify this segment if the LS bit in the ENV register is 1 and the CS bit is 0.

5 System Library. This map defines one of up to 32 additional segments for operating system code. These segments--absolute segments %34 (#28) and up--may be viewed as an extension to the system code segment and are common to all processes. Map 5 maps the absolute segment invoked by the most recent call instruction (XCAL or DPCL) or return instruction (EXIT or IXIT) to or from these segments. All code references (except "UC" mode addresses) specify this segment if the LS bit in the ENV register is 1 and the CS bit is 1.

6-13 I/O Buffers and Segment Page Tables. Buffers for I/O transfers and the Segment Page Tables are normally kept in the segments defined by these maps. They are always associated with absolute segments 6 through 13, respectively. The Segment Page Tables and the use of I/O buffers are discussed later in this section.

14 Special-Purpose Area. This map is not used to map any entire segment, but is reserved by the system for special purposes. It is divided into several areas:

Area	Map Entries
Microcode Scratch Registers	0:27
Map Entries for Segment Table (SEG)	28:43
Map Entries for Physical Page Segment Table (PHYSEG)	44:51
Map Entries for Physical Page/Logical Page Table (PHYPAGE)	52:59
Base for Current Extended Data Segment	60:61
Limit for Current Extended Data Segment	62:63

The scratch registers are for use by the processor microcode. The SEG, PHYSEG, and PHYPAGE tables are used for mapping and other memory management functions; their map entries reside permanently in Map 14, and the tables themselves reside in absolute segment 14. The current base and limit, which are used in resolution of relative extended addresses in segments 4 and up, are also stored here for efficiency.

15 Extended Address Cache. This map is not used to map any segment, but is used for the extended address cache (discussed later in this section).

The segments mapped by Maps 0 through 5 (short address spaces) are accessible by 16-bit addressing and by relative extended addressing as relative segments 0 through 3. The absolute segments mapped by maps 0, 2, and 4 change as different processes come into execution, since new sets of code and data are mapped by the "user" maps. When a process is activated, Map 0 is loaded with the entries that define the process's data space, and Map 2 is loaded with the entries for the current segment of the process's code space. Map 4 is loaded with the entries for one segment of the process's library code space, if any, on demand (by XCAL, DPCL, EXIT, or IXIT instructions).

On the other hand, Maps 1, 3, and 5 do not change when different processes are dispatched. Maps 1 and 3 always map the same absolute segments; Map 5 changes to another absolute segment only when a call to a system procedure references a procedure that is in a system library segment other than the one currently mapped.

At any given time, each segment mapped by maps 0 through 14 corresponds to some specific absolute segment. This correspondence is maintained in a software table called CSSEG (Current Short-address Segments), which has entries for all 16 maps.

For relative extended addressing of the segments represented by Maps 0 through 5, four segments are accessible as relative segments 0 through 3 (refer back to Figure 5-7). For efficiency, memory access for these segments normally uses the maps, rather

than the extended address translation algorithm. The map used for a given relative segment number is determined by the settings of the LS, PRIV, DS, and CS bits in the ENV register. The currently accessible segments, with their relative segment numbers, are defined as follows:

0 Current Data Segment. If DS=0, map 0 (user data) is used to access this segment; if DS=1, map 1 (system data) is used. This specifies the same segment that a LOAD G+0 instruction would access.

1 System Data Segment. If DS=1, or if explicit reference is made to system data (e.g., with SG-relative addressing) and PRIV=1, map 1 (system data) is used to access this segment. Otherwise, map 0 (user data) is used. This specifies the same segment that a LOAD SG+0 instruction would access.

2 Current Code Segment. If LS=0 and CS=0, map 2 (user code) is used to access this segment. If LS=0 and CS=1, map 3 (system code) is used. If LS=1 and CS=0, map 4 (user library) is used. If LS=1 and CS=1, map 5 (system library) is used. This specifies the same segment that instructions are fetched from and that an LWP instruction would access.

3 Currently Mapped User Code Segment. Map 2 is always used to access this segment. This specifies the same segment that an LWUC instruction would access.

Map Entries and Mapping

As already mentioned, each map contains 64 map registers. Each map register in maps 0 through 13 (and in parts of map 14 and map 15) contains a map entry. Map entries are used to convert logical addresses to physical addresses.

In the case of 16-bit addressing or relative extended addressing in segments 0 through 3, the map is first selected; it is always one of Maps 0 through 5. Then the logical page number from the 16-bit address or the 32-bit relative extended address is used as an index into the map to obtain the map entry.

Figure 5-11 shows the format of a map entry. Since maps are loaded from Segment Page Tables, this format also applies to entries in Segment Page Tables and in the map entry cache, both of which are described later.

If bit 15 is not set, bits 0 through 12 of the map entry indicate the physical page number (0 through 8191) of the memory page to be accessed whenever a memory reference is made through this

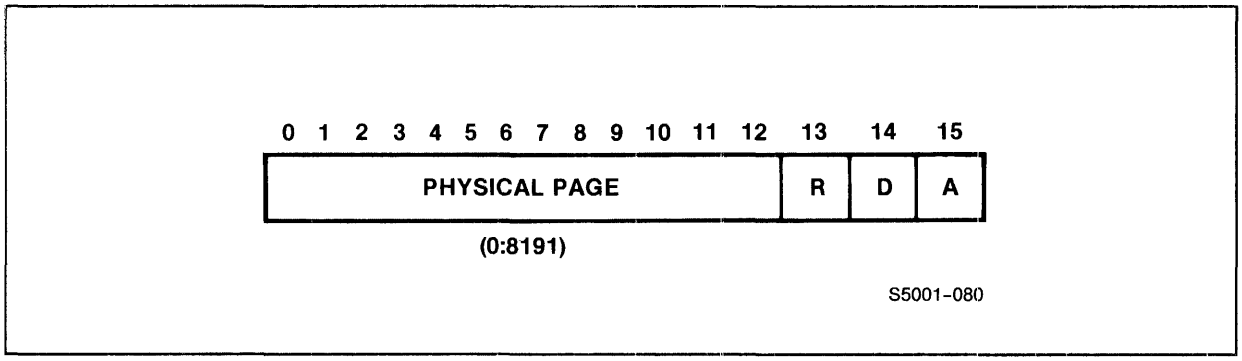


Figure 5-11. Map Entry

entry. (If bit 15 is set, the page is considered absent from physical memory; all other bits are undefined to the hardware, although the memory manager process may use these bits.) Bit 13, the R (reference) bit, is set on any access to the page. Bit 14, the D (dirty) bit, is set whenever a write access is made to the page. These two bits are checked by the memory manager software in the operating system in order to select the best pages for overlay when absent pages need to be brought into physical memory from disc, and to keep track of whether a page that is being replaced must first be copied to disc (i.e., is a dirty page). Bit 15, the A (absent) bit, if set to 1, indicates that the page referred to is not present in physical memory. An attempt to access memory via an entry with this bit set to 1 will result in a Page Fault interrupt if the attempted access was made by an instruction, or a transfer error if the I/O channel attempted the access.

Once the map entry is selected, bit 15 of the entry is checked to determine if the page is absent. If so, a page fault interrupt occurs, and the page fault interrupt handler takes over to swap in the page from disc. Once the physical page is present, the physical page field of the map entry (now updated) is used to select it, and the word field of the 16-bit or 32-bit address is used to select one of the 1024 words within the page. See Figure 5-12.

Segment Table and Segment Page Tables

Pages accessed by 16-bit addresses, or by relative addresses with segment numbers of 0 through 3, are usually already mapped at the time they are referenced by a procedure. A map entry (in Maps 0 through 5) provides the physical location of the page. However,

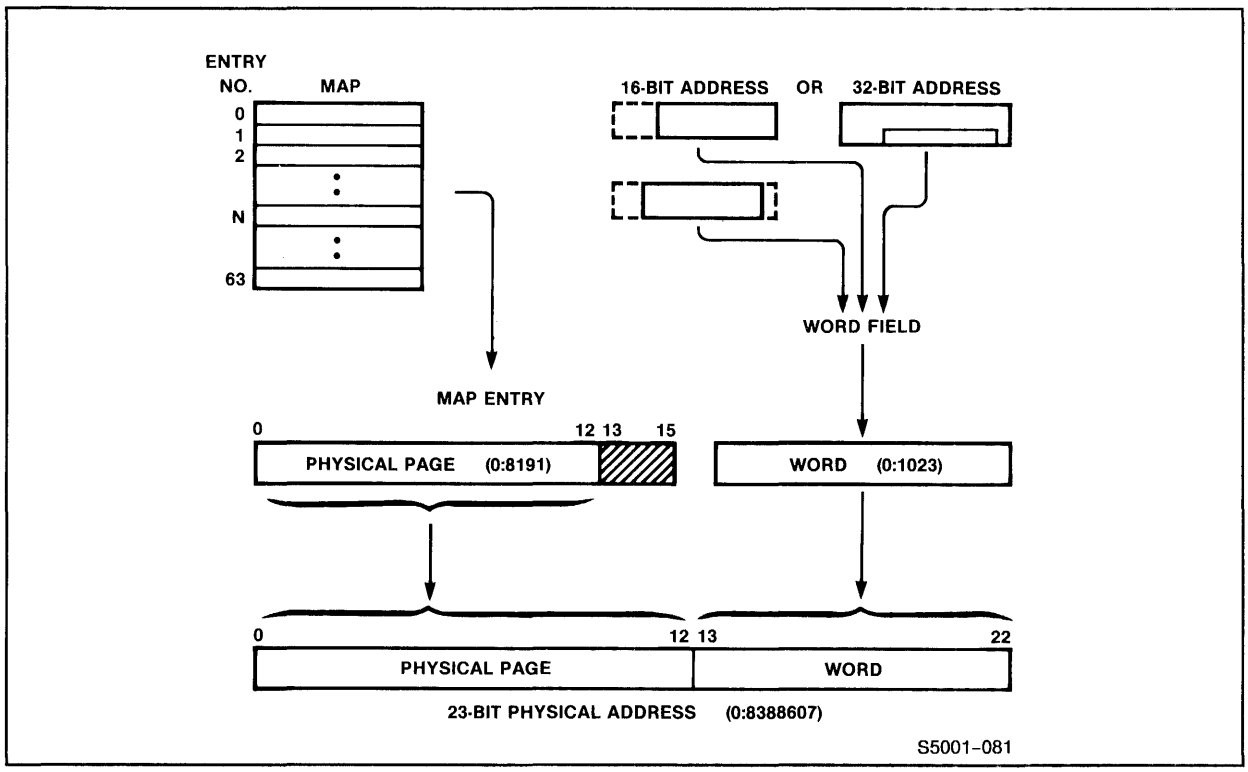


Figure 5-12. Mapping

pages referenced by calls to procedures in segments other than those currently mapped, or by absolute extended addresses, or by relative extended addresses with segment numbers of 4 or greater, are not necessarily already mapped at the time the address reference is made.

Before any page can be accessed, its map entry must exist in a map. The processor maintains sets of tables in memory so that the appropriate map entry for any page in a process's logical memory can be located and brought into a map when needed. These tables are permanently mapped so that the processor can always access them through a normal mapped reference.

The Segment Table contains a two-word entry for each absolute segment of the CPU's virtual memory. Each allocated segment entry (some segment numbers may not be allocated) points to the Segment Page Table for that segment, and indicates whether the segment is mapped. See Figure 5-13.

There is one Segment Page Table (SPT) for each allocated absolute segment (Figure 5-13). Each Segment Page Table contains a map

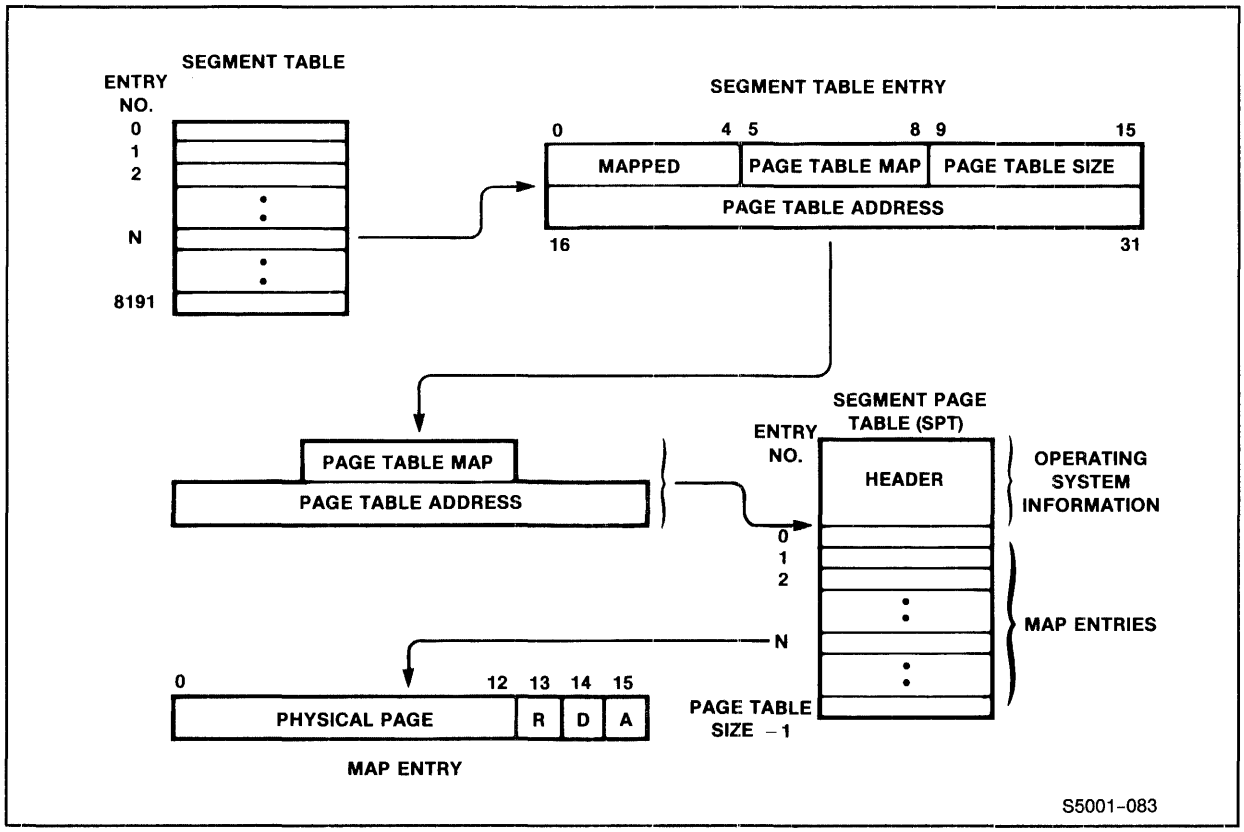


Figure 5-13. Segment Table and Segment Page Tables

entry for each allocated page in the corresponding segment. (Informally, an SPT is frequently called a "page table.")

The Segment Table has a two-word entry for each absolute segment. Thus it occupies 16k words (16 pages) of physical memory. It is permanently mapped in entries 28 through 43 of Map 14. Bits 0 through 4 of a Segment Table entry contain the map number of the map for that segment if the segment is currently mapped (in this case, all other bits of the entry can be ignored), or all ones (%37) if the segment is not mapped. Bits 5 through 8 specify which map (by convention, in the range 6 through 13) maps the Segment Page Table for that segment.

Bits 9 through 15 specify the number of pages (0 to 64) in the segment; this equals the size of the Segment Page Table in words, not including the header. The remainder of the entry (the second word) gives the 16-bit address of the Segment Page Table for the segment. This address, together with the map number in bits 5 through 8, specifies the location of the Segment Page Table in memory.

Segment Page Tables for segments that are currently allocated but not in use (i.e., not mapped) are located in an area of memory called MAPPOOL. The operating system allocates the MAPPOOL areas out of absolute segments 6 through 13, which are permanently mapped by Maps 6 through 13. In MAPPOOL, each Segment Page Table is preceded by a header that gives operating system information, and it contains only as many entries as there are pages in the segment.

The Segment Page Table for each allocated absolute segment contains one entry for each page in the segment. Each of these entries, identical to map entries (Figure 5-11), defines the physical memory where a page of the segment resides, or indicates that the page is absent from physical memory. If a segment is not allocated (i.e., the page table size entry in the Segment Table is equal to 0), then no Segment Page Table exists for that segment.

When a new process is dispatched by the operating system, the entries in the Segment Page Table for that segment are copied into a map using the MAPS instruction. (When this is done, if fewer than 64 pages of the segment are allocated, the remainder of the entries in the map are marked absent by setting these entries equal to 1.)

For data segments, if the segment being addressed is currently mapped, the only valid copy of the map entries is the one in the map; the copy kept in the Segment Page Table in memory is updated only when the segment is unmapped. For code segments, map copy and the memory copy are generally identical.

Extended Address Cache

For a memory access using an extended address, the address translation algorithm requires that the Segment Table entry for the required absolute segment be examined to determine whether the segment is mapped. If the segment is not mapped, then it becomes necessary to use the extended address cache, which occupies all of Map 15.

The extended address cache (see Figure 5-14) is used in memory accesses that specify absolute extended addresses (or relative extended addresses with relative segment numbers of 4 or greater) to temporarily map one single page being referenced. Its content is a collection of page mappings for recently accessed pages, which thus have a high probability of being accessed repeatedly in succeeding references. This greatly improves the speed of access to frequently used pages.

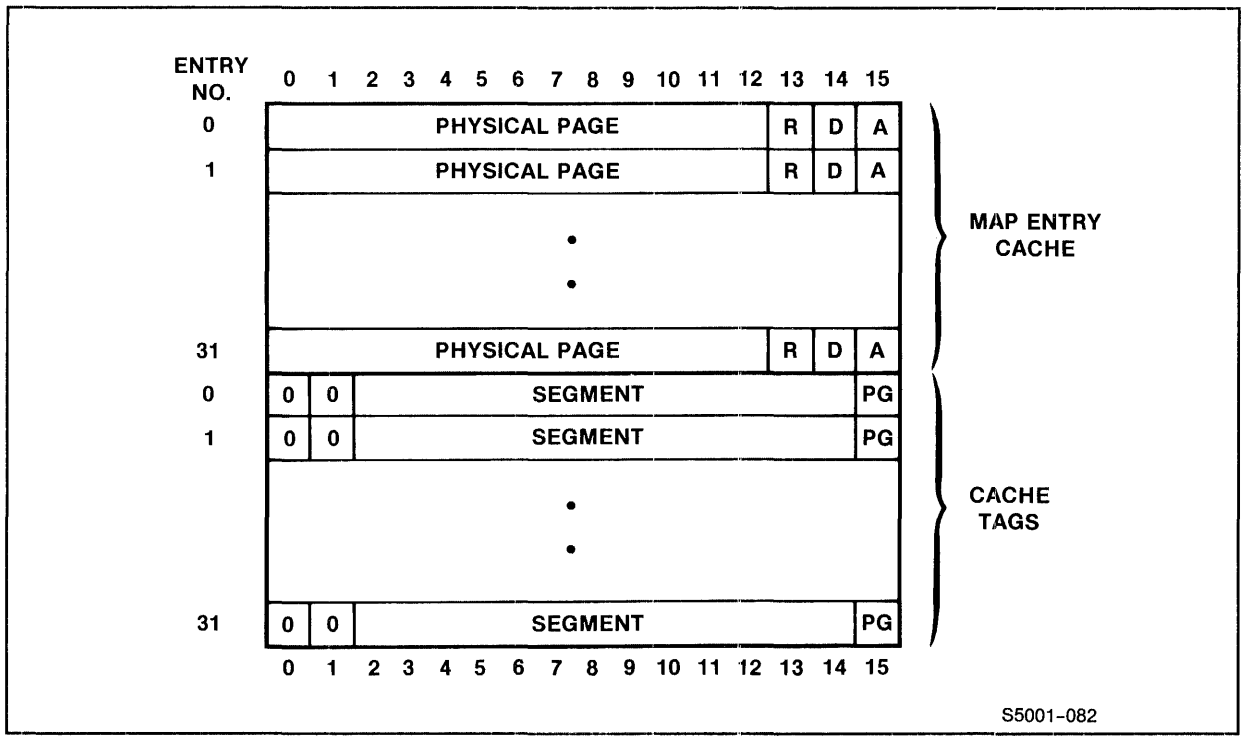


Figure 5-14. Extended Address Cache

As shown in the figure, the cache is divided into two halves: the map entry cache and the cache tags. The map entry cache consists of 32 map entries; these are identical in format to the map entries in the other maps and in the Segment Page Tables in memory. The 32 cache tags each contain a 13-bit segment number and a single bit that represents the most significant bit of a page number. A cache tag identifies the corresponding entry in the map entry cache.

Using the extended address cache, a byte represented by an absolute extended address is accessed as follows (see Figure 5-15):

1. The corresponding cache tag is obtained by using the least significant 5 bits of the page number from the address as an index into the entries in the upper half of Map 15.

2. The cache tag is compared to the high-order word of the absolute extended address (ignoring bits 0 and 1 of the extended address). If they are equal, the correct map entry is present at the corresponding position in the map entry cache. The map entry is obtained by using the least significant 5 bits of the page number from the address as an

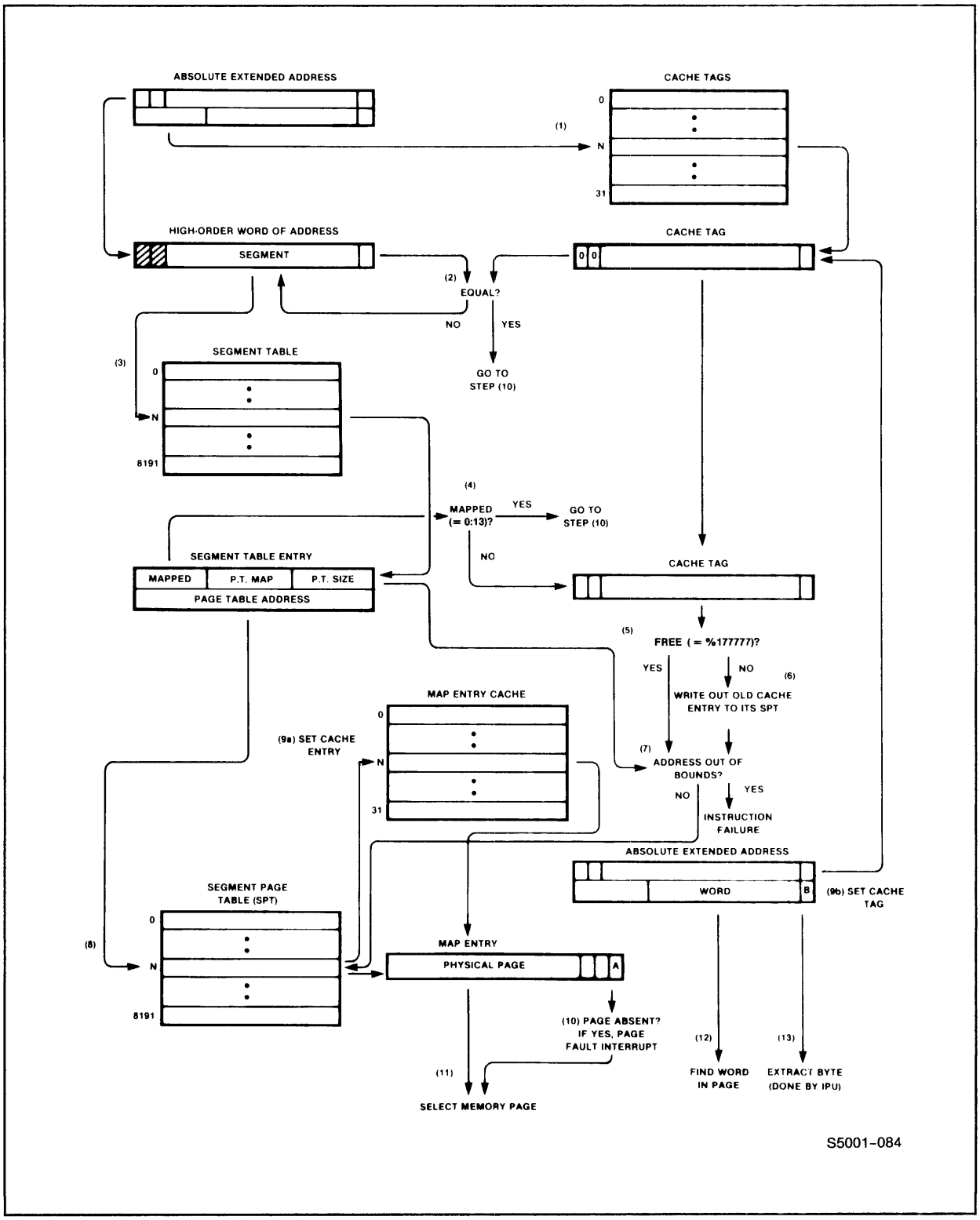


Figure 5-15. Extended Address Translation Algorithm

index into the lower half of Map 15. The page in memory can then be accessed using the map entry (go to step 10).

3. If the desired map entry is not in the cache, the absolute segment number from the address is used as an index into the Segment Table to find the appropriate entry for this segment.

4. Bits 0 through 4 of the entry are checked to determine whether the segment is currently mapped. If bits 0 through 4 are not all ones (%37), the segment is currently mapped, and the map number is given in this field. The page number from the address is used as an index into the specified map to find the entry for the desired page. The page in memory can then be accessed using the map entry (go to step 10).

5. If bits 0 through 4 of the Segment Table entry are all ones, the segment is not currently mapped. The cache tag is then checked to see if the corresponding cache entry is free (not in use); a free cache entry is indicated if the tag is equal to -1 (%177777). If the entry is free, go to step 7.

6. If the entry is not free, the existing cache entry is written out to its corresponding Segment Page Table entry in memory. This is done by using the information in the existing cache tag and cache entry to go through the appropriate Segment Table entry.

7. Bits 9 through 15 of the Segment Table entry for the new address (the page table size field) are compared with the page number in the address. If the page number is greater than or equal to the number of pages given in the Segment Table, the address is out of bounds, and an instruction failure trap occurs.

8. The page table map and page table address in the Segment Table entry are used to find the beginning of that segment's Segment Page Table in memory. Then the page number from the absolute extended address is used as an index into the Segment Page Table to find the entry for the desired page.

9. The appropriate map entry in the cache is loaded with the Segment Page Table entry, and the corresponding cache tag is set to match the high-order word of the absolute extended address (ignoring bits 0 and 1).

10. If the map entry shows that the page is absent, a page fault interrupt occurs. The page fault interrupt handler then takes over to swap in the desired page from disc, as discussed under "Page Fault" later in this section.

11. Once the page is in main memory, the physical page number found in the map entry is used to select the physical page.

12. The word field of the address is used to locate the desired word in memory.

13. The byte field of the address can then be used by the IPU to extract the specified byte.

MEMORY ACCESS (NonStop TXP PROCESSOR)

This subsection describes the actual mechanisms used to access memory in the NonStop TXP processor. This information is needed primarily by systems analysts, though it may also be of interest to others. The information parallels that of the preceding subsection (describing memory access for the NonStop II processor).

Short Address Spaces

A process accesses logical memory either by 16-bit logical addresses or by 32-bit relative extended addresses.

The 16-bit addresses access one of six address spaces, called "short address spaces" (SASs). These six address spaces constitute a process's normal view of memory. The SASs are known as: user data, system data, user code, system code, user library, and system library.

Some of these six short address spaces can consist of more than one logical segment, and each such segment corresponds to a specific absolute segment. At any given time, only one segment of a short address space is the currently "mapped" segment. In a NonStop TXP processor, the absolute segment number of the currently mapped segment for each SAS is kept in the first six locations of a set of hardware registers called the SST (Short Segment Table); a copy of the SST is kept in a software table, CSSEG (the Current Short-address Segment table). However, unlike the NonStop II, the NonStop TXP processor normally does not fully map all 64 pages of a current segment, but rather "caches" individual page mappings when needed. (The exceptions are SAS 1 and SAS 3, system data and system code, which are always fully mapped in PCACHE--described later.)

The current segments of the six short address spaces are defined as follows:

0 User Data Segment. This is the data segment for the process that is currently in execution. If DS is a 0, all data references will be into this segment unless they are by instructions which use either extended addressing or the SG addressing mode.

1 System Data Segment. This segment, which is always absolute segment 1, contains system tables and interrupt stacks. This segment is common to all processes, but it may only be accessed if DS or PRIV is set.

2 User Code Segment. This is the current segment of the user code space for the currently executing process; that is, it is the absolute segment invoked by the most recent call instruction (XCAL or DPCL) or return instruction (EXIT or (IXIT) to or from these segments. All code segment references specify this space if the CS and LS bits in the ENV register are 0. In addition, "UC" mode addresses always reference this segment regardless of the ENV register bit settings.

3 System Code Segment. This segment, which is always absolute segment 3, contains the most frequently used operating system procedures and interrupt handlers. This segment is common to all processes. All code references (except "UC" mode addresses) specify this segment if the LS bit in the ENV register is 0 and the CS bit is 1.

4 User Library Segment. This is the current segment of the user library space for the currently executing process. That is, it is the absolute segment (if any) invoked by the most recent call instruction (XCAL or DPCL) or return instruction (EXIT or IXIT) to or from these segments. User library segments are mapped "on demand"; until there is a library call, none of these segments is current. All code references (except "UC" mode addresses) specify this space if the LS bit in the ENV register is 1 and the CS bit is 0.

5 System Library Segment. This is the current segment of the system library, which provides additional code space for system procedures. These segments--absolute segments %34 (#28) and up--be viewed as an extension to the system code segment (SAS 3) and are common to all processes. SAS 5 is the absolute segment invoked by the most recent call instruction (XCAL or DPCL) or return instruction (EXIT or IXIT) to or from these segments. All code references (except "UC" mode addresses) specify this space if the LS bit in the ENV register is 1 and the CS bit is 1.

Other segments defined by the SST are the following:

6-13 Buffers and Tables. Buffers for I/O transfers and Segment Page Tables are normally kept in these spaces, which are absolute segment numbers 6-13, respectively.

14 Memory Management. This space is absolute segment 14. It is divided into several areas:

Reserved for microcode (scratch)	%0:%067777
Segment Table (SEG)	%70000:%127777
Physical page Segment table (PHYSEG)	%130000:%147777
Physical page Page table (PHYPAGE)	%150000:%167777
Reserved for microcode (scratch)	%170000:%177777

The portions of this space marked "Reserved for microcode" are not available to the GUARDIAN operating system because the resources that would be used to access them in a NonStop II processor have been allocated to its micro machine.

In a NonStop II processor, slots 0:%33 of map 14 are scratch registers and slots %74:%77 hold the extended base and extended limit for the current extended data segment. Therefore, in the NonStop TXP processor, the corresponding page table cache entries (for pages 0:%33 and %74:%77 of absolute segment 14) may be used as scratch registers. The page table cache (PCACHE) is described later in this section.

15 Unused by the GUARDIAN operating system. The NonStop TXP processor microcode may use the SST register for short address space 15 as a scratch register. (The NonStop II processor uses map 15 for the extended address cache.)

Caches in the NonStop TXP Procesor

The NonStop TXP processor gains much of its performance through the use of cache memory. Cache memory is a mechanism used to improve effective memory transfer rates and increase processor speed. The term "cache" refers to the fact that a copy of frequently used information is cached close to where it will be used--on the processor logic boards. The cache mechanism is essentially hidden and is transparent to the user.

Processors that do not use cache memory need to go to physical memory for every word accessed. Cache memory allows the system to store frequently used information in a set of hardware registers to take advantage of fast register access time as compared to slower memory access time.

When a process requests information that it needs to continue its work, the processor microcode first checks whether that information is present in cache. The NonStop TXP processor uses two caches to speed access to memory.

- PCACHE is a page table cache that stores frequently used page table entries. Its use is analogous to that of the map registers in the NonStop II processor.

- CACHE is a cache that stores frequently used blocks of code or data in another set of hardware registers. Transfers between memory and CACHE occur by single 16-byte fetch operations.

Both caches are accessed by absolute extended addresses. Because both caches are much smaller than the processor's virtual address space of one gigabyte, it is possible for multiple logical addresses to map into a given cache entry. However, in the cache technique chosen for the NonStop TXP processor, a tag word associated with each cache entry isolates the entry to a unique logical address. This type of cache is commonly referred to as "direct-mapped" or "single-set associative" cache.

To ensure that information in main memory remains consistent with the version in cache, the processor updates the information in main memory for each write to cache. This technique is commonly known as "write-through."

The following paragraphs describe the operation of both caches.

Page Table Cache (PCACHE). Unlike the NonStop II processor, the NonStop TXP processor does not have sixteen sets of map registers. Instead, the NonStop TXP processor uses a 2048-location hardware register array called "PCACHE" to store frequently used page table entries.

In a directly-mapped cache, each possible entry maps into exactly one cache location. In the case of PCACHE, "each possible entry" is each combination of absolute segment and page within a segment. Because there are many more potential entries than cache locations, it is necessary to associate a tag with each cache location to identify which entry is in that cache location at a given moment. The tags for PCACHE are kept in another 2048-location hardware array called PCACHETAG.

PCACHE is divided into two halves. The first half is a "dedicated" cache which continuously maps the page table entries for absolute segments 0-15. This ensures that the page table entries of the physical memory pages assigned to system data, system code, I/O buffers, and memory management will always be available--they will never be swapped out.

The second half of PCACHE is managed as a direct-mapped or single-set associative cache. Its 1024 locations store individual page table entries related to absolute segments other than segments 0-15. The technique of storing individual page table entries eliminates the overhead of loading the page table entries for an entire segment into registers when access is required to only a single page.

The hardware maintains the segregation of the dedicated and single-set associative sections of PCACHE--they appear to the microcode and the software to be a single cache.

PCACHE is accessed directly by 32-bit extended addresses. This means that each absolute extended address uniquely maps into a single cache location. The layouts for both PCACHE and PCACHETAG are shown in Figure 5-16.

The microcode executes the following four operations each time that it services a request for memory access through PCACHE:

1. reads a PCACHE entry

2. determines whether or not the entry is valid

3. generates a physical memory address

4. reports on the status of the memory page designated in the physical memory address generated.

The function that maps an address into PCACHE is:

```
(segment.<0:8><>0) ^ segment.<9:12> ^ page.<0:5>
```

which can also be thought of as the concatenation of a bit indicating whether the segment number is in the range of 0:15 with bits 11:20 of the extended address. It is the checking of segment.<0:8> that causes the partitioning of PCACHE. If all nine bits are 0, the segment number must be <=15, and the mapping is forced into the first half of PCACHE; if any of those nine bits is nonzero, the address mapping is forced into the second (single-set) half of PCACHE. The remaining information needed to uniquely identify the entry (segment.<0:8>, that is, extended address bits 2:10) is kept in the entry's associated tag.

The notation used to reference a Segment Page Table entry is:

PCACHE[segment, page]=	segment page table entry for an absolute segment (0-8191) and logical page (0-63)
PCACHETAG[segment, page]=	tag for that segment and page's entry

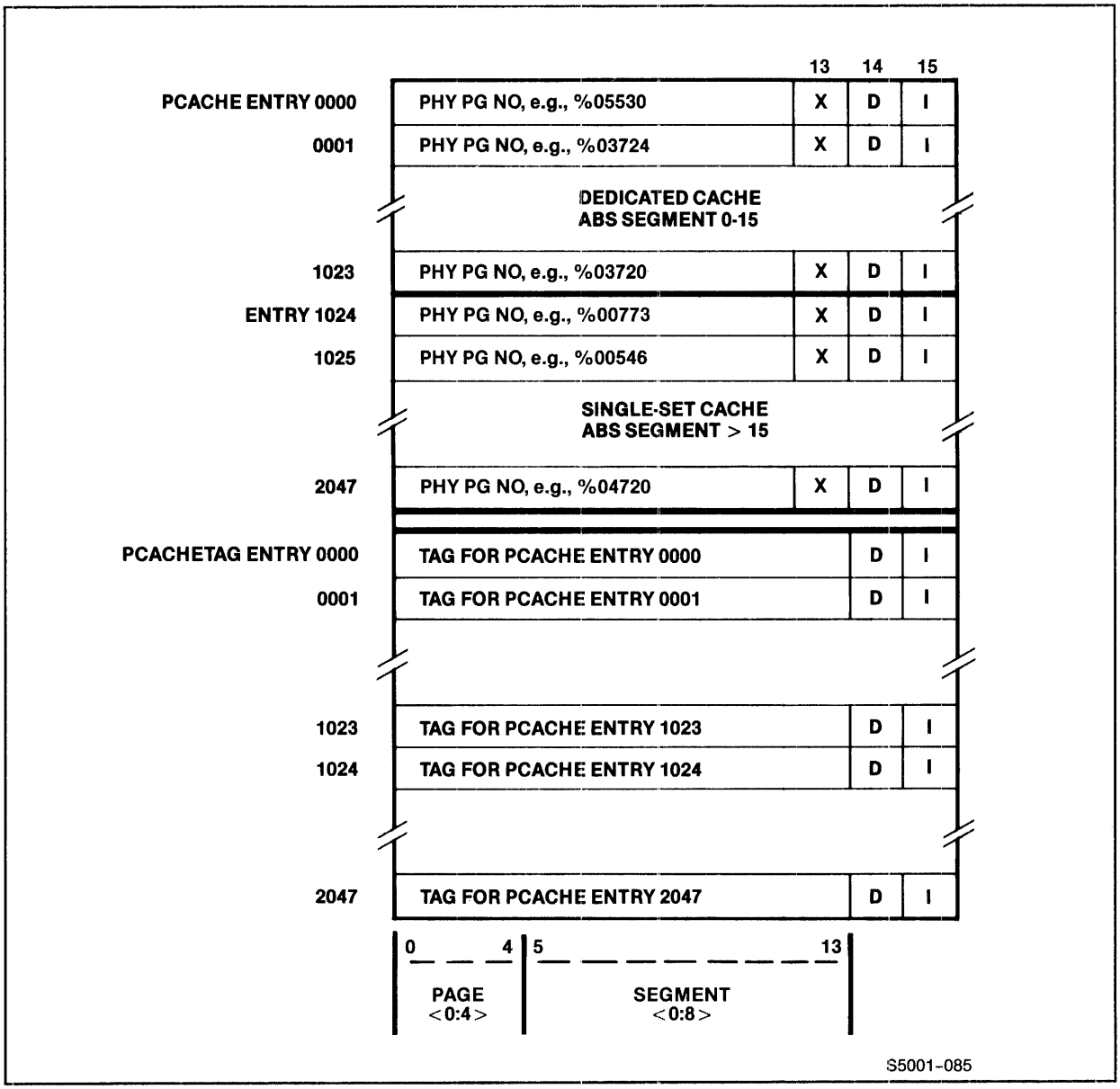


Figure 5-16. Layout of PCACHE

where PCACHE contains a physical page number and PCACHETAG contains the corresponding tag, in the formats shown in Figure 5-16. (The PCACHE Segment field equals zero in entries for absolute segments 0 through 15.) The PCACHETAG "Page" field is a copy of the upper five bits of the physical page number given in PCACHE; this is used by the microcode for preprocessing before the full physical page number is available in PCACHE.

Bits 13:15 of PCACHE contain the following status information:

PCACHE.<13> is the reference bit. In the context of PCACHE, this bit can be interpreted as a "don't care" bit because it is assumed to be set to 1 for any valid entry.

PCACHE.<14> and its copy in PCACHETAG.<14> act as the dirty bit. If this bit is set to 1, a data page has been modified. This means the memory manager must swap this page to disc before it can give away that page's physical memory space.

PCACHE.<15> and its copy in PCACHETAG.<15> serve as the invalid bit. This bit is 1 if the page is absent or the entry is invalid. If the invalid bit is 1, the system then checks the absent bit in the memory-resident copy of the Page Table. If it is also set to 1, then the page is considered to be absent.

Data Cache. The NonStop TXP processor maintains a 64K-byte cache that holds a combination of instructions and data. This "code-and-data" cache, called CACHE, is a direct-mapped or single-set associative cache. It provides parallel, high-speed access to pieces of data stored in physical memory. (For this discussion, the term "data" applies to either an instruction or an operand.) The layout of CACHE "data store" and its associated "tag store" is shown in Figure 5-17.

Like PCACHE, CACHE is accessed by 32-bit absolute extended address. This means that before CACHE can be checked for the presence of a desired word, the address of that word must be in absolute extended address format. Address conversion for the three kinds of addresses (absolute extended, relative extended, and "short address") is handled in the manner shown below:

- An absolute extended address requires no conversion.
- A relative extended address must be converted to an absolute address. That is, the system adds a base address offset to the relative address and performs a bounds check on the resultant absolute address.
- A "short address" must be converted to an absolute address. A short address is a 16-bit address (logical page within the segment and word offset within a page) combined with selected bits of the ENV register.

All references to memory are routed through CACHE. CACHE executes the following four operations every time it services a memory reference request:

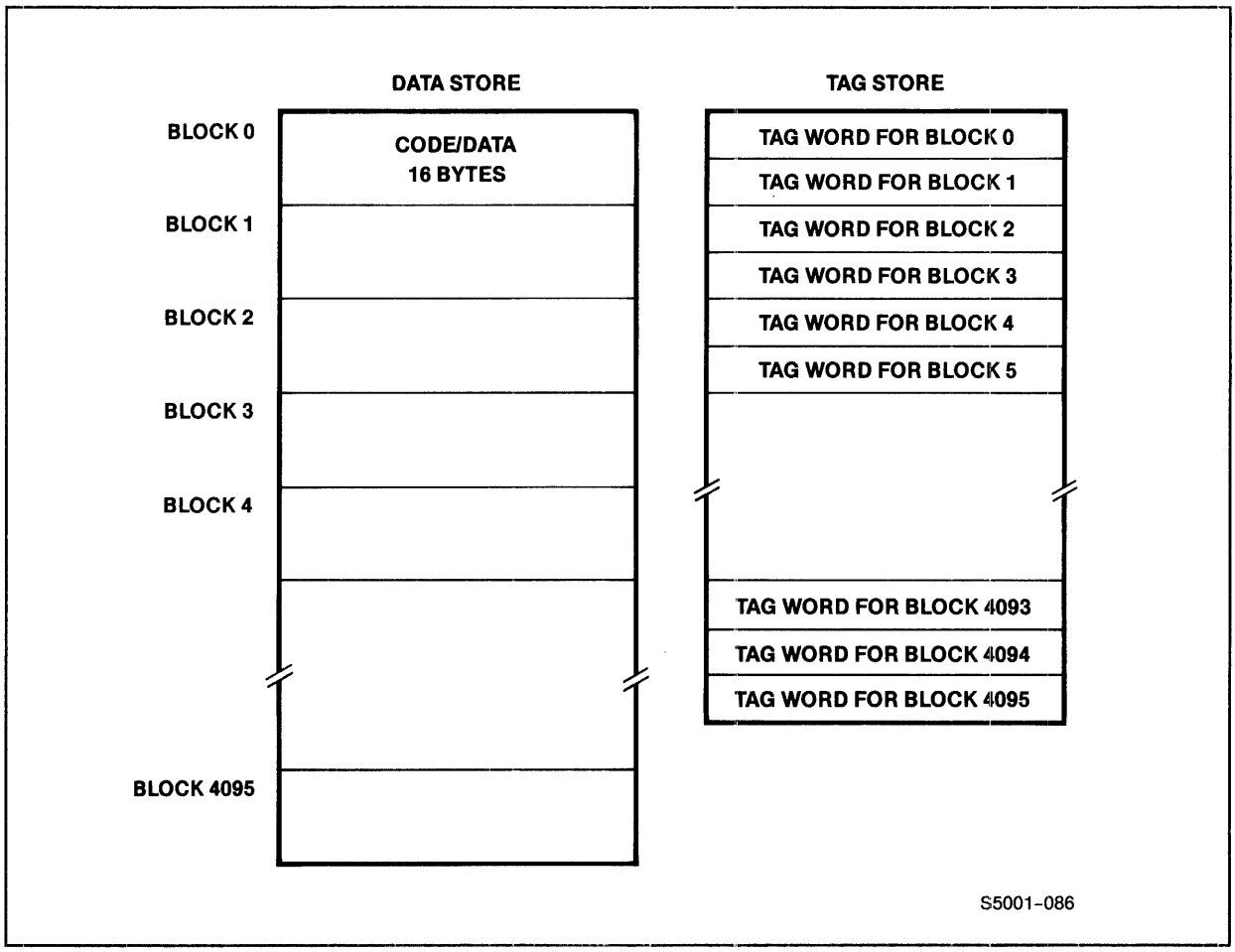


Figure 5-17. Layout of CACHE

1. retrieves code or data from CACHE

2. generates a CACHE fault if the code/data is invalid

3. generates a physical memory address in anticipation of the next operation

4. reports on the status of the memory page designated in the physical memory address generated.

CACHE is arranged in 16-byte blocks whose starting addresses fall on 16-byte boundaries. This minimizes the number of tags required (one for each 16-byte block) and also speeds up the filling of cache. Figure 5-18 shows a simplified view of how CACHE is accessed.

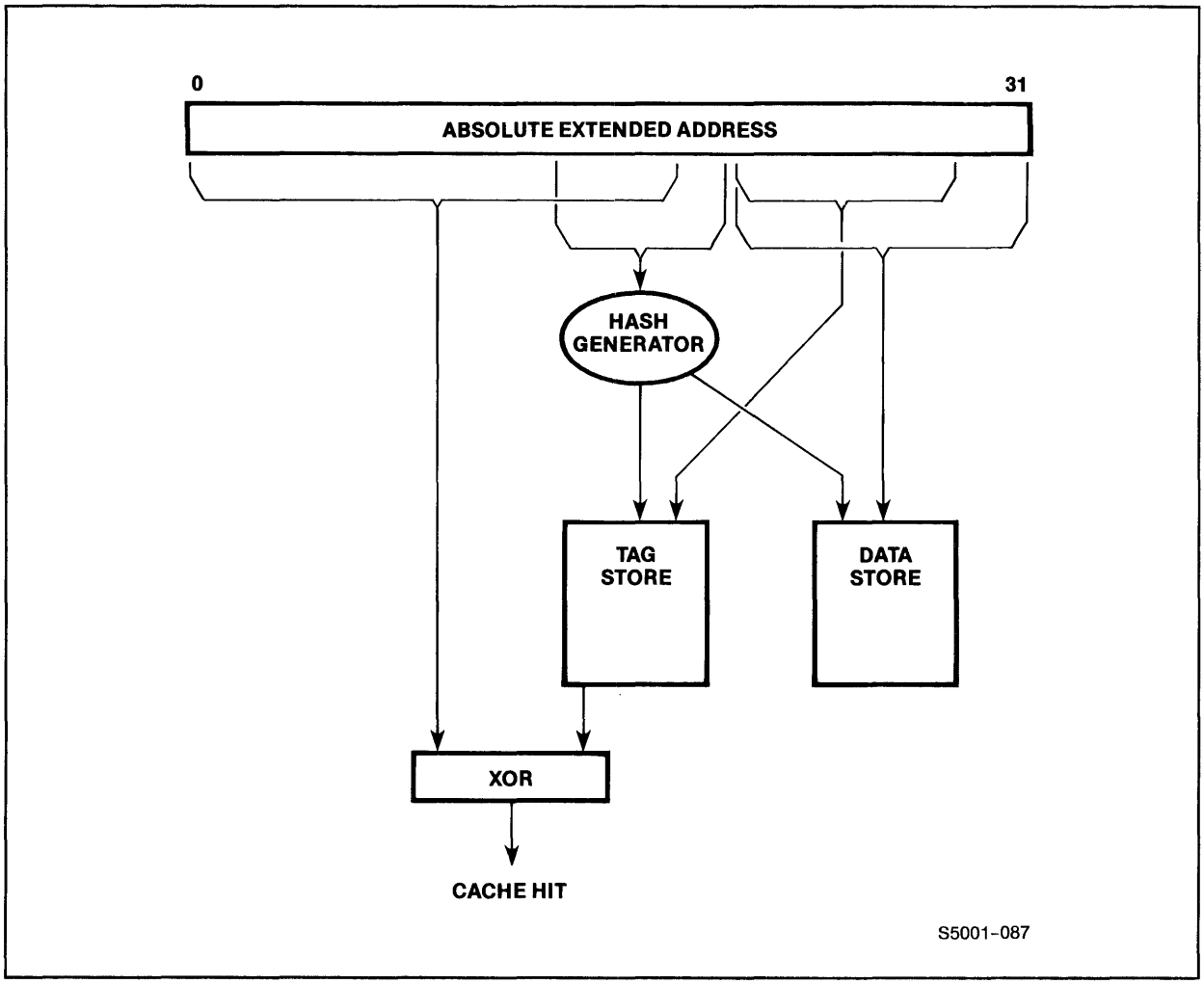


Figure 5-18. Access to CACHE

As indicated, the hardware applies a hashing function to the extended address to evenly distribute highly accessed areas of CACHE; e.g., System Data segment page 0. Once the data has been brought into cache by the initial "fault", the hashing algorithm assures a high probability that the requested code or data will already be in cache the next time that it is needed.

The hardware performs an XOR (exclusive OR) operation on the tag and selected bits of the 32-bit extended address. The result of the XOR (i.e., cache hit or miss) indicates whether or not that block's contents correspond to the address used. If a cache miss occurs, the current code or data is discarded and the intended code or data is "faulted in" from memory.

MEMORY DATA STRUCTURES

Several data structures known by the hardware and maintained by the operating system play an active role in performing memory management tasks. Briefly, these system data structures include:

- SEG (Segment Table) resides in segment 14, words %70000:%127777.
- Segment Page Tables are scattered through segments 6-13.
- CSSEG (Current Short-address Segment table) resides in memory mapped by segment 1, words %1340:%1357. This table maintains the correlation between address spaces 0:15 and their associated absolute segments.
- SST (Short Segment Table) resides in hardware registers--for NonStop TXP processors only. It contains a copy of the CSSEG table contents.
- PHYSEG (Physical page Segment table) resides in segment 14, words %130000:%147777.
- PHYPAGE (Physical page Page table) resides in segment 14, words %150000:%167777.

The Segment Table and Segment Page Tables define whether or not a segment is mapped, and if mapped the physical memory it occupies.

The NonStop II processor uses only CSSEG to maintain the correlation between address spaces 0:15 and specific absolute segments. Each entry either contains a value in the range 0:%17777 for segments 0:8191 or a -1 (i.e., no segment is currently in this short address space). The NonStop TXP processor normally uses the register-speed path through the SST in preference to the slower path through the memory-resident CSSEG table.

The memory manager process handles requests for individual pages by searching the PHYPAGE and PHYSEG tables to see whether or not the page is available. The PHYPAGE table contains a one-word entry for each page of physical memory. It is accessed by physical page number index; i.e., entries 0:%17777 correspond to physical pages 0:8191. Each PHYPAGE table entry contains the following information:

- PHYPAGE[p].<10:15> contains a given segment's logical page number 0:%77.

- PHYPAGE[p].<0:9> is set to zero when the physical page is allocated. This bit-field is subsequently available to the memory manager for recording additional "usage" information.

Correspondingly, the PHYSEG table contains a one-word entry for each physical page of main memory. It too is accessed by physical page number index; i.e., entries 0:%17777 for physical pages 0:8191. Each PHYSEG table entry contains one of the following items:

- %0 <= PHYSEG[p] <= %17777 indicates that page 'p' is in segment PHYSEG[p] and the page may be swapped out. When the memory manager must select a page that is already allocated to another process, it uses the PHYSEG table to locate the associated segment number and then flags the page as "absent" in that segment's Page Table.

- PHYSEG[p] = %40000 indicates that page 'p' is free. This means the page is not currently allocated to any process and is available for overlay.

- PHYSEG[p] = %40001 indicates that page 'p' has had an UCME (uncorrectable memory error) and is no longer available.

- PHYSEG[p] = %40002 indicates that page 'p' has had a hard CME (a single-bit error that can be corrected but causes system interrupts on every reference) and is no longer available.

- PHYSEG < 0 indicates that page 'p' is locked into memory and cannot be swapped out. -PHYSEG[p] is the number of locks queued on the page.

I/O ADDRESSING

The memory mapped by address spaces 6 through 13 (i.e., absolute segments 6 through 13) represents one megabyte of logical address space. This space is accessible only by using absolute extended addresses; however, it is a special case because it is always fully mapped. As a result, memory accesses to these segments are fairly fast, because they need not go through the Segment Page Tables. These segments may be accessed using only 20 bits of information--a 4-bit absolute segment number, a 6-bit page number to locate the entry within the segment, and a 10-bit word offset. Absolute segments 6 through 13 are used by the operating system for two purposes: for MAPPOOL storage, which (as has already been discussed) contains the Segment Page Tables, and for I/O buffers. Because these segments are reserved for operating system use, only privileged processes (such as I/O processes) can

access them. (Being accessible only by absolute extended addresses provides this protection.)

The I/O channel addresses its buffers by means of the I/O Control (IOC) table, which is located in page 1 of the system data segment. Fields within the IOC entry for the subchannel associated with a device keep track of the channel's current position in the buffer during a transfer.

Before beginning a transfer, the I/O process initializes the IOC entry. The segment base, page base, and page offset fields are initialized with the segment number, page number, and word offset for the beginning of the buffer. The segment number, of course, is always in the range 6 through 13. The byte count field is initialized with the total number of bytes to be transferred (which can be odd or even), and the segment offset field is initially zero. I/O buffers do not need to begin on page boundaries, and they may span page boundaries and segemnt boundaries; however, they always begin on a word boundary.

As the transfer proceeds, the third word of the IOC entry--containing the page offset field in the low-order bits and the segment offset in the high-order bits--is incremented and the byte count is decremented. The segment base and page base fields in the IOC entry remain unchanged, but the segment and page numbers of the word to be accessed at any given time are obtained by adding the segment offset to the page base, using any overflow to increment the segment number. The transfer continues until the byte count is zero or an I/O error occurs. See Figure 5-19.

The NonStop TXP processor caches active IOC entries. This cache, as well as the operation of the I/O channel, is described more fully in Section 7.

PAGE FAULT

A page fault occurs when a reference is made to a page that does not currently reside in main memory. The absent page can be a code page, a data page that has been previously written into and then swapped out to disc ("dirty" page), a new data page containing initialization data that must be read in from disc, or a data page with no initialized or previously written data ("clean" page).

When a page fault is detected, an interrupt to the operating system page fault interrupt handler occurs. The following discussion assumes familiarity with the hardware mechanism for handling interrupts, as described under "Interrupt System" in Section 6.

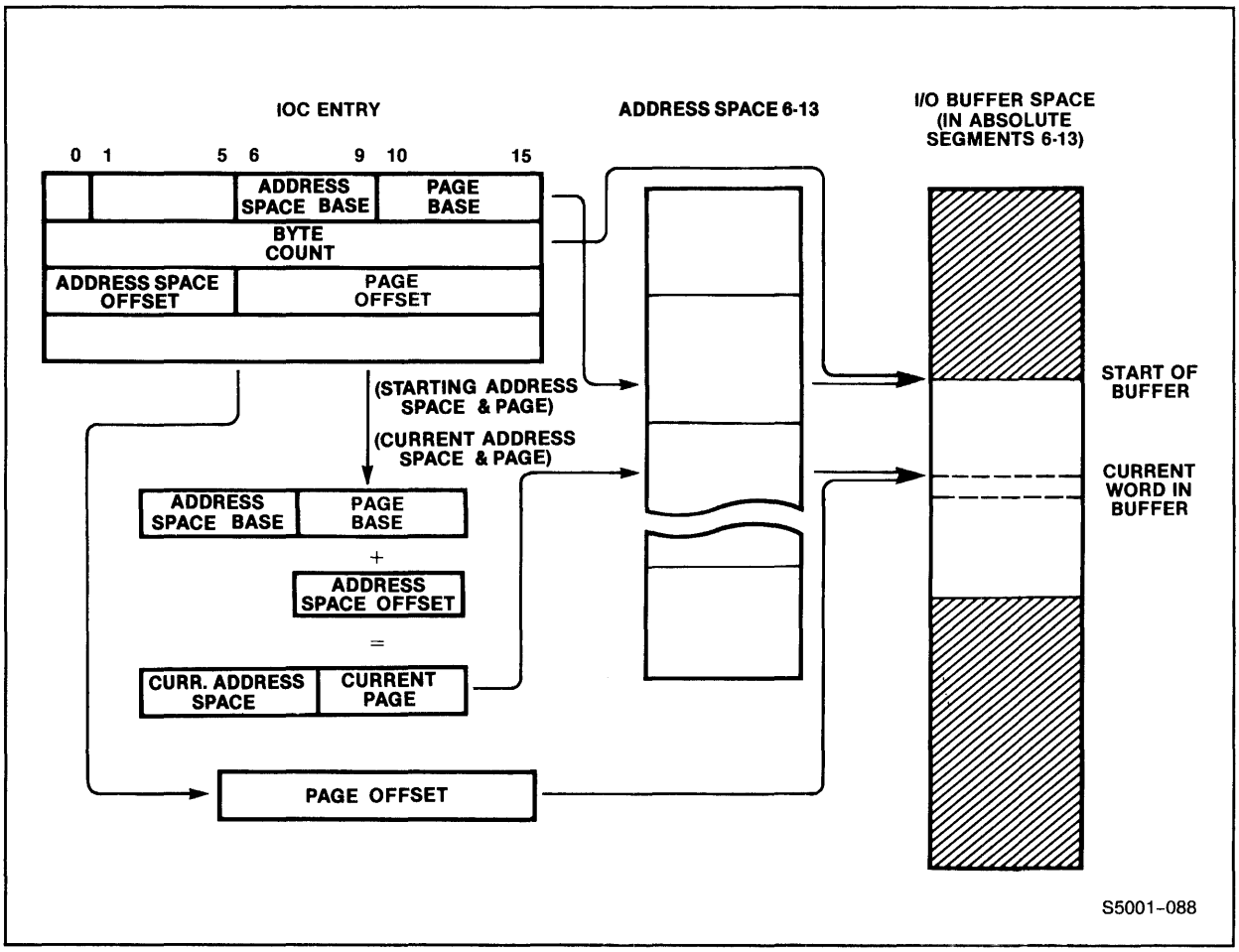


Figure 5-19. I/O Buffer Addressing

The page fault interrupt sequence is illustrated in Figures 5-20a and 5-20b, which show an example page fault for a user data page. The sequence is as follows:

1. An address reference is made to a page that is absent from physical memory; that is, a page whose entry has its A (absent) bit set to 1.

2. An interrupt through the System Interrupt Vector (SIV) table entry at SG[%1220]--the entry for the page fault interrupt--occurs. The hardware passes the absolute extended address of the absent page to the interrupt handler. (For a NonStop TXP processor, the low-order word of the address contains the word offset within the page; for the NonStop II processor it is cleared to zero.) The high-order word of the address is passed as the interrupt parameter in the Vi location of the

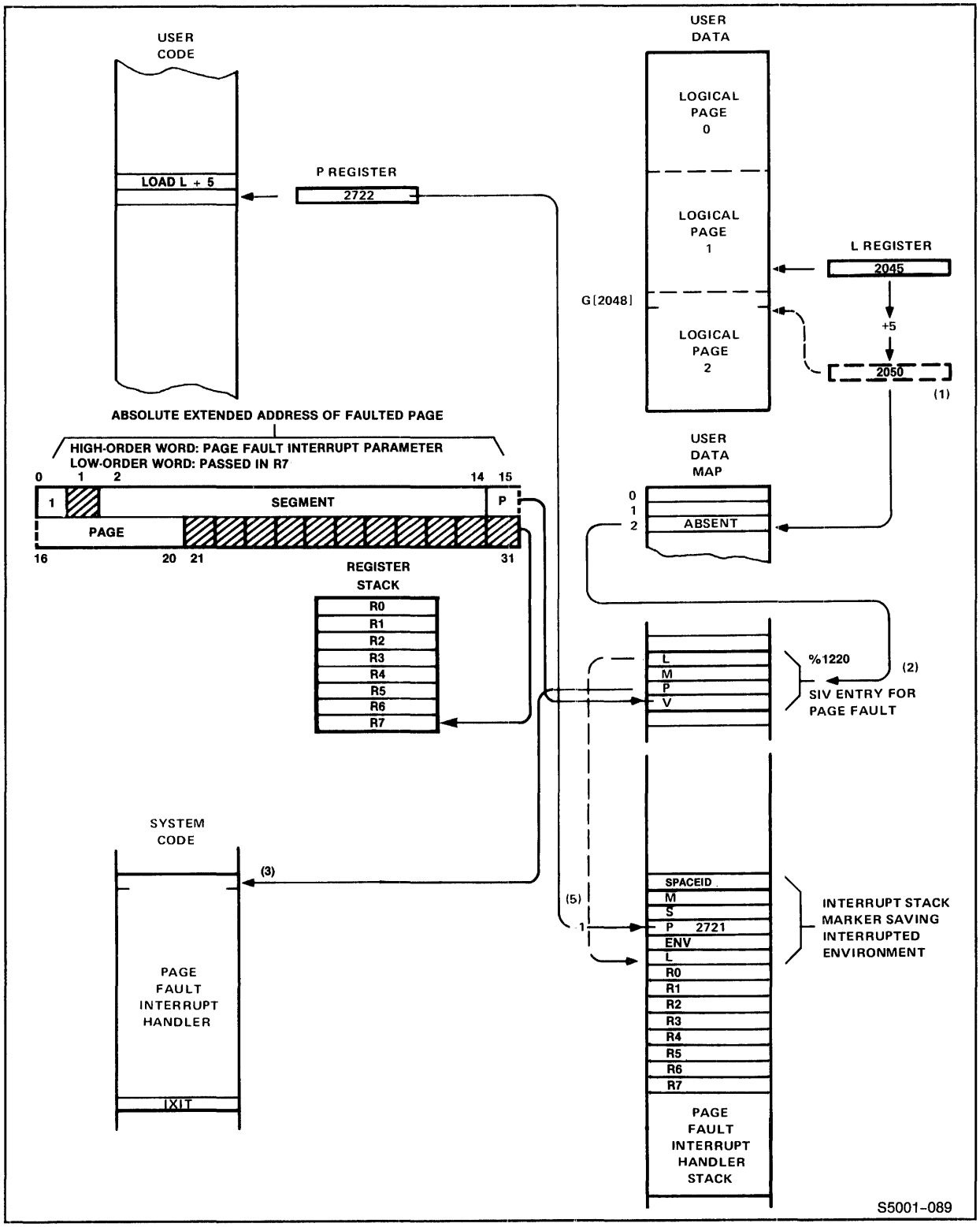


Figure 5-20a. Page Fault Interrupt Sequence

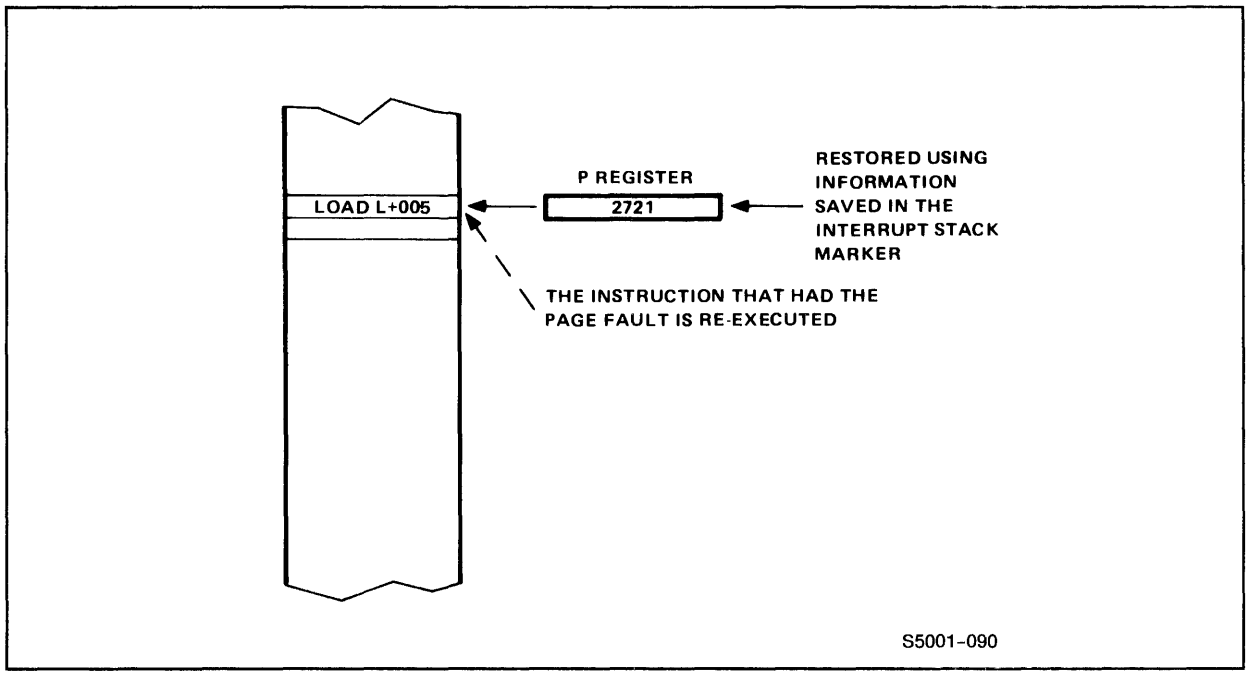


Figure 5-20b. Page Fault Interrupt Sequence

SIV entry. The low-order word of the address is passed in R7 of the Register Stack after the current environment has been saved.

The current P Register setting is decremented by one (so that the faulted instruction will be repeated upon return from the interrupt handler), and then the current environment--Space ID, interrupt mask (M), the S, P, ENV, and L Registers, and the Register Stack--is saved in the interrupt stack marker. The interrupt environment is established in the manner described later in Section 6.

3. The page fault interrupt handler saves the interrupted environment and the absolute extended address of the absent page, and passes control to the Dispatcher.

4. The Dispatcher in turn invokes the memory manager process. If necessary to make room in physical memory for the new page, the memory manager chooses another page already in main memory and removes (or "replaces") it. The memory manager reads the absent page from disc, overlaying the replaced page, and then sets the page table entry for the retrieved page to the address of its physical page. The process is then allowed to execute again.

The memory manager can replace only "clean" pages. These are either code pages or pages for which the "dirty" bit is not set. The memory manager periodically cleans dirty pages by writing them to disc and clearing the dirty bit.

5. Because the P Register setting of the faulted environment was decremented by one before it was saved, the instruction previously causing the page fault is now reexecuted.

MEMORY ERRORS

Correctable and uncorrectable memory errors are reported to the processor either as interrupts or as I/O termination conditions. An uncorrectable error generally indicates that the physical page should no longer be used. A correctable error, on the other hand, may occur because of either a transient failure or a hard error. A hard error can be detected by rewriting a page that gets a correctable error and then seeing if the error occurs again. A privileged instruction, CMRW, is used by the operating system for this purpose; this instruction holds off memory accesses by the I/O channel while a word of memory is being rewritten.

SYSTEM TABLES

The locations of some major tables discussed at length later in this manual are illustrated in Figure 5-21. These tables are located in pages 0 and 1 of the system data segment, which are always located in physical pages 0 and 1, respectively. Note that all of page 1 is used for the I/O Control Table (IOC).

The following paragraphs briefly describe the tables shown in Figure 5-21.

System Interrupt Vector. SG[%1200:%1337] is the System Interrupt Vector (SIV). This table contains 24 four-word entries; each entry defines the executing environment for one of the operating system interrupt handlers (see "Interrupt System," Section 6).

Bus Receive Table. SG[%1400:%1477] is the Bus Receive Table (BRT). This table contains 16 four-word entries, each of which is assigned to manage the interprocessor bus transfers for one

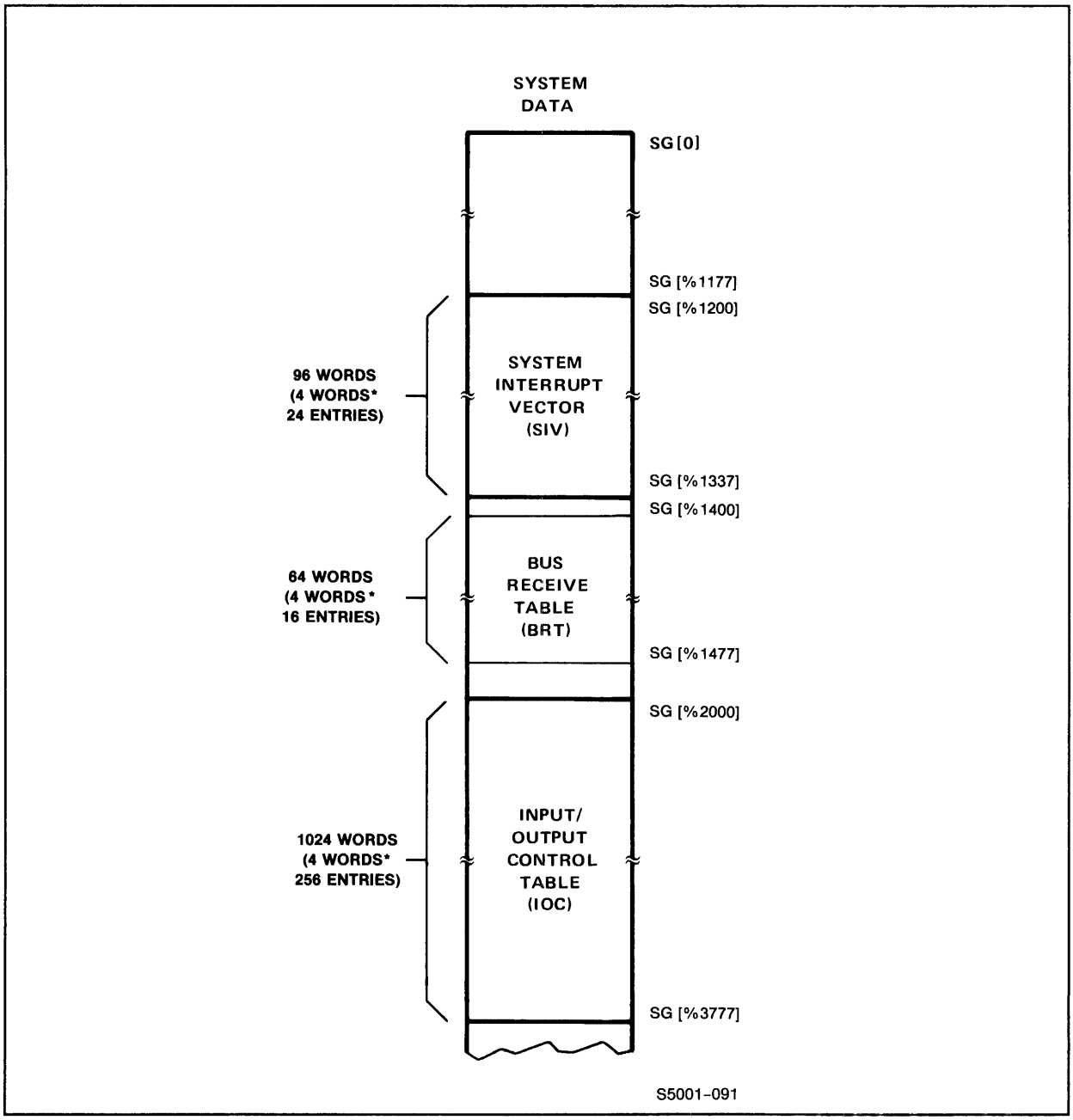


Figure 5-21. Dedicated Memory Locations in System Data

processor module. Each entry describes the number of words expected and the system buffer location where the data is to be stored (see "Interprocessor Buses" in Section 7).

Bus Receive Table Long. SG[%1600:%1677] is the Bus Receive Table Long (BRTLONG). This table contains 16 four-word entries, each of which points to the BRT entries for another cluster in the FOX network (see "Interprocessor Buses" in Section 7).

I/O Control Table. SG[%2000:%3777] is the I/O Control Table (IOC). This table contains 256 entries corresponding to the 256 subchannels that can be connected to an I/O channel. Each entry describes the number of bytes to be transferred and the system buffer location to be used for the data transfer (see "Input/Output Channel" in Section 7).

SECTION 6

INTERRUPT SYSTEM

The interrupt system transfers control to a specific location in the operating system (called an interrupt handler) upon the occurrence of any of the conditions listed in Table 6-1. All interrupt handlers for these events are located in the system code segment (SAS 3, or absolute segment 3).

When an interrupt occurs, the interrupted environment is saved in an interrupt stack marker. An operating system interrupt handler executes to process the particular interrupt. Then an IXIT (interrupt exit) instruction is executed to restore the interrupted environment (see Figure 6-1).

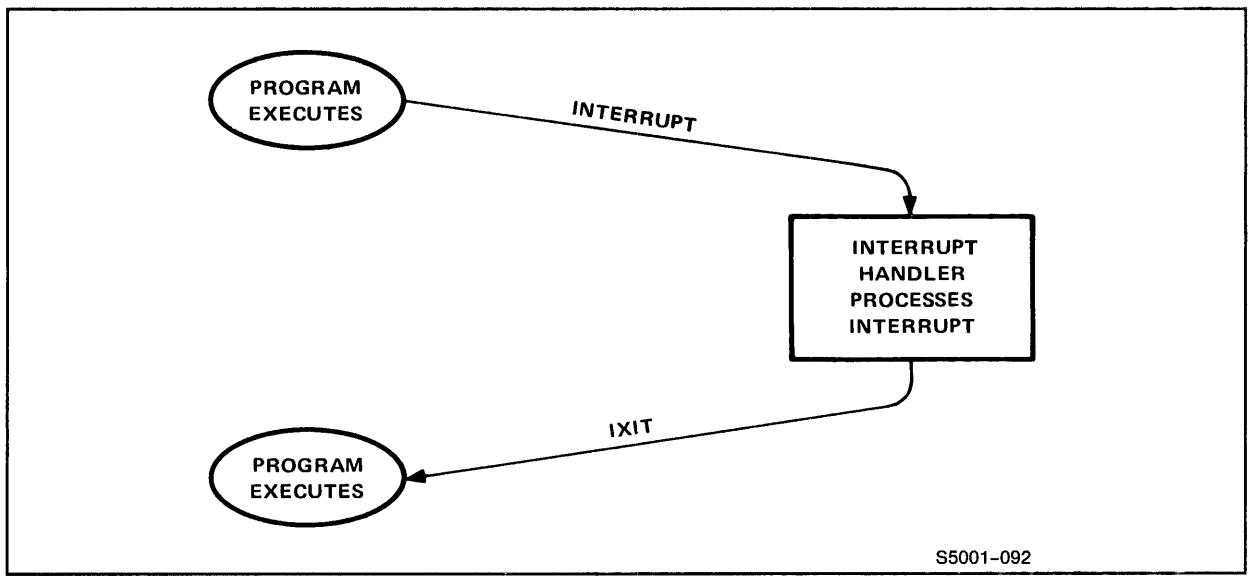


Figure 6-1. General Interrupt Sequence

Table 6-1. Interrupt Conditions

Interrupt No.	Event
0	Special channel error
1	Uncorrectable memory error
2	Memory access breakpoint
3	Instruction failure
4	Page fault
5	Undefined
6	Undefined
7	OSP (Operations & Service Processor) I/O
8	Power fail
9	Correctable memory error
10	High-priority I/O
11	Interprocessor bus receive completion
12	Undefined
13	Time list
14	Standard I/O
15	Dispatcher
16	Power on
17	Stack overflow
18	Arithmetic overflow or divide by zero
19	Instruction breakpoint
20	XRAY Sampler (NonStop TXP processor only)
21-23	Undefined

INT AND MASK REGISTERS

Three registers are associated with interrupts: two 16-bit interrupt registers (INTA and INTB) and a 16-bit MASK register. The bit assignments of these registers are illustrated in Figure 6-2. Only four bits of INTB are relevant to interrupts; however, these four are the highest-priority interrupt bits, being examined first at the conclusion of each instruction. The interrupts represented by the bits of INTA are maskable--that is, the corresponding bits of the MASK register are used by the operating system to allow or disallow particular interrupt types at various critical or noncritical times. Bit 6 of INTA (arithmetic overflow or divide by zero) is separately masked by the trap enable bit of the Environment Register (ENV.<8>), but is used in a similar way to enable or disable that interrupt. For all maskable interrupts, the interrupt condition is ignored if the corresponding MASK bit is equal to 0, and will continue to be deferred until the MASK bit is set to 1. The checking operation is performed by a logical AND of the two registers.

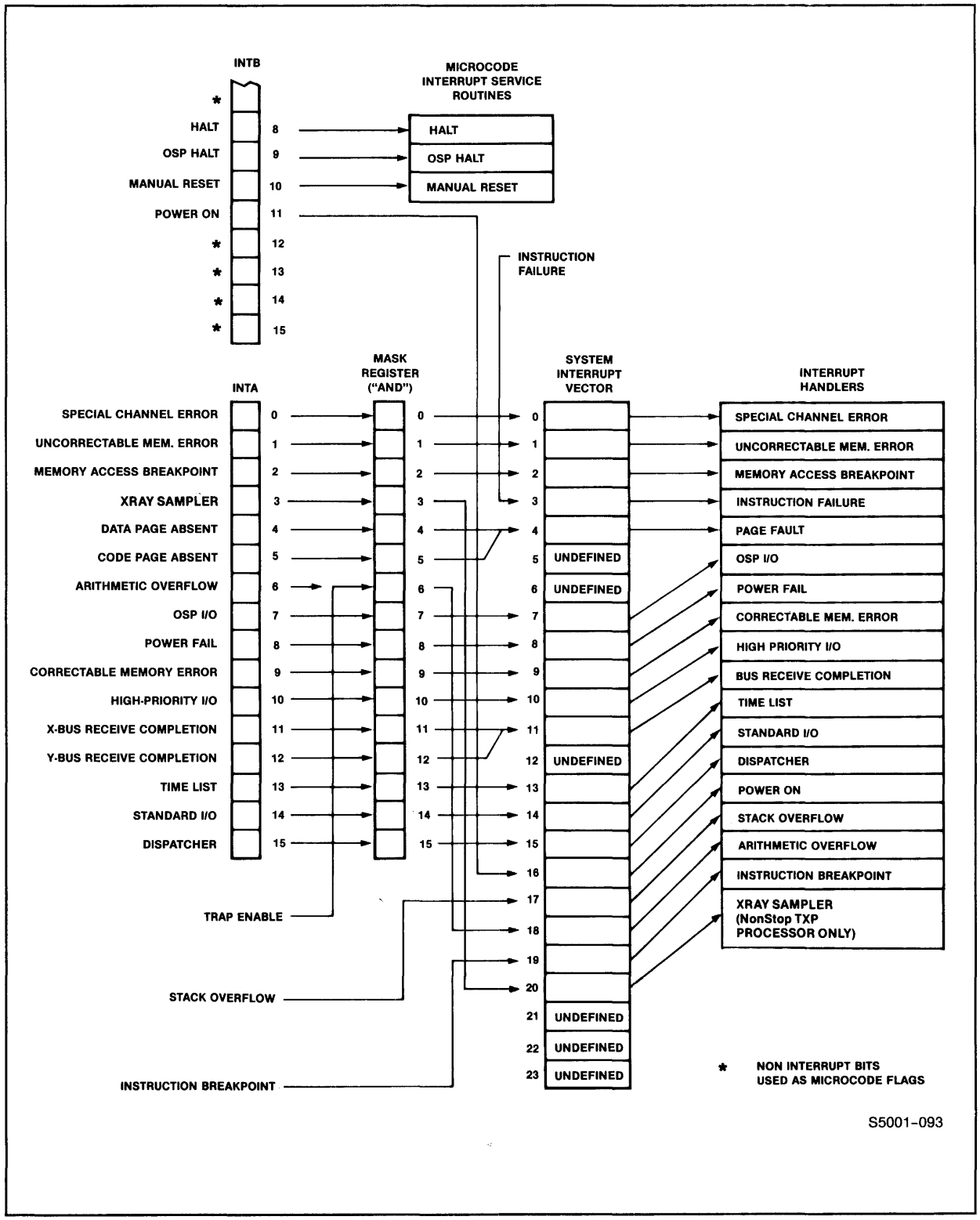


Figure 6-2. INT and MASK Registers

Most interrupt types can occur only at the end of an instruction, when the hardware routinely checks for the presence of 1 bits in the interrupt registers. However, three interrupt types (power on, uncorrectable memory error, and page fault) are preemptive; that is, they will interrupt during an executing instruction. Also, certain long-running instructions (e.g., the move instructions) may be interrupted during execution.

If two or more interrupt conditions exist simultaneously in INTA, and each has its corresponding MASK register bit set, the interrupt type with the highest priority (lowest bit number) takes precedence; the others are deferred until the interrupt handler finishes executing and executes an IXIT instruction.

Interrupts for stack overflow, instruction failure, and instruction breakpoint have entries neither in the interrupt registers nor in the MASK register; these cause an interrupt whenever they occur, ignoring priority. The hardware-only interrupts (halt, OSP halt, and manual reset) are serviced entirely within microcode.

As shown in Figure 6-2, detected interrupt conditions are passed to software interrupt handlers through the System Interrupt Vector.

SYSTEM INTERRUPT VECTOR

Each interrupt event that is to be serviced by software has a corresponding entry in the System Interrupt Vector (SIV). The SIV, which is initialized by the operating system, defines the executing environment for each of the 18 operating system interrupt handlers. The SIV, shown in Figure 6-3, begins at system data location %1200 and contains 24 four-word entries (six are undefined).

Each four-word entry in the System Interrupt Vector contains the following information:

Li = L register setting for interrupt handler
Mi = MASK register setting for interrupt handler
Pi = P Register setting of first instruction in interrupt handler
Vi = Interrupt-related parameter put here by firmware

The following paragraphs further describe the functions of each of these entries, as illustrated in Figure 6-4.

- Li: This is the address in system data space for an interrupt handler's local storage (stack).

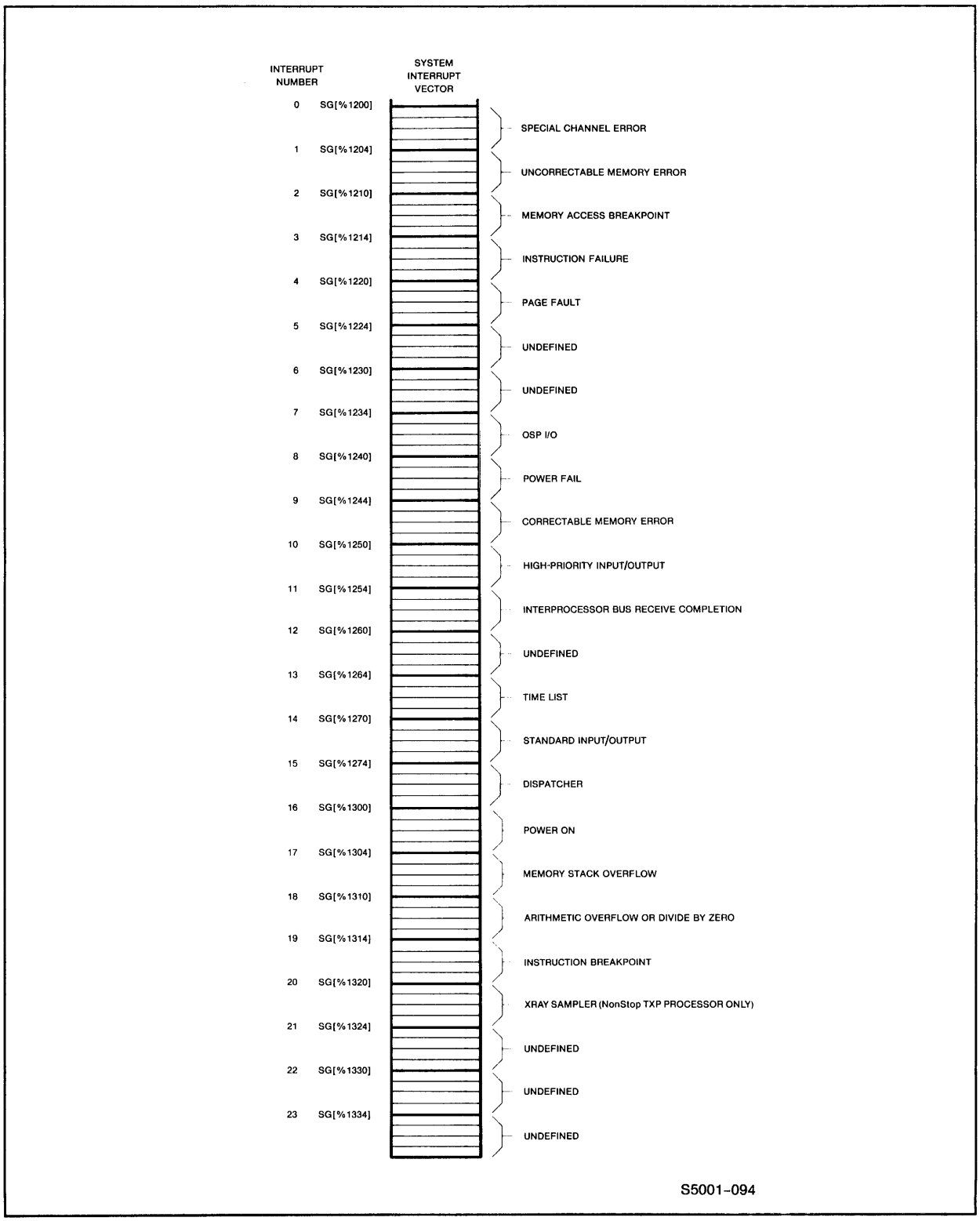


Figure 6-3. System Interrupt Vector

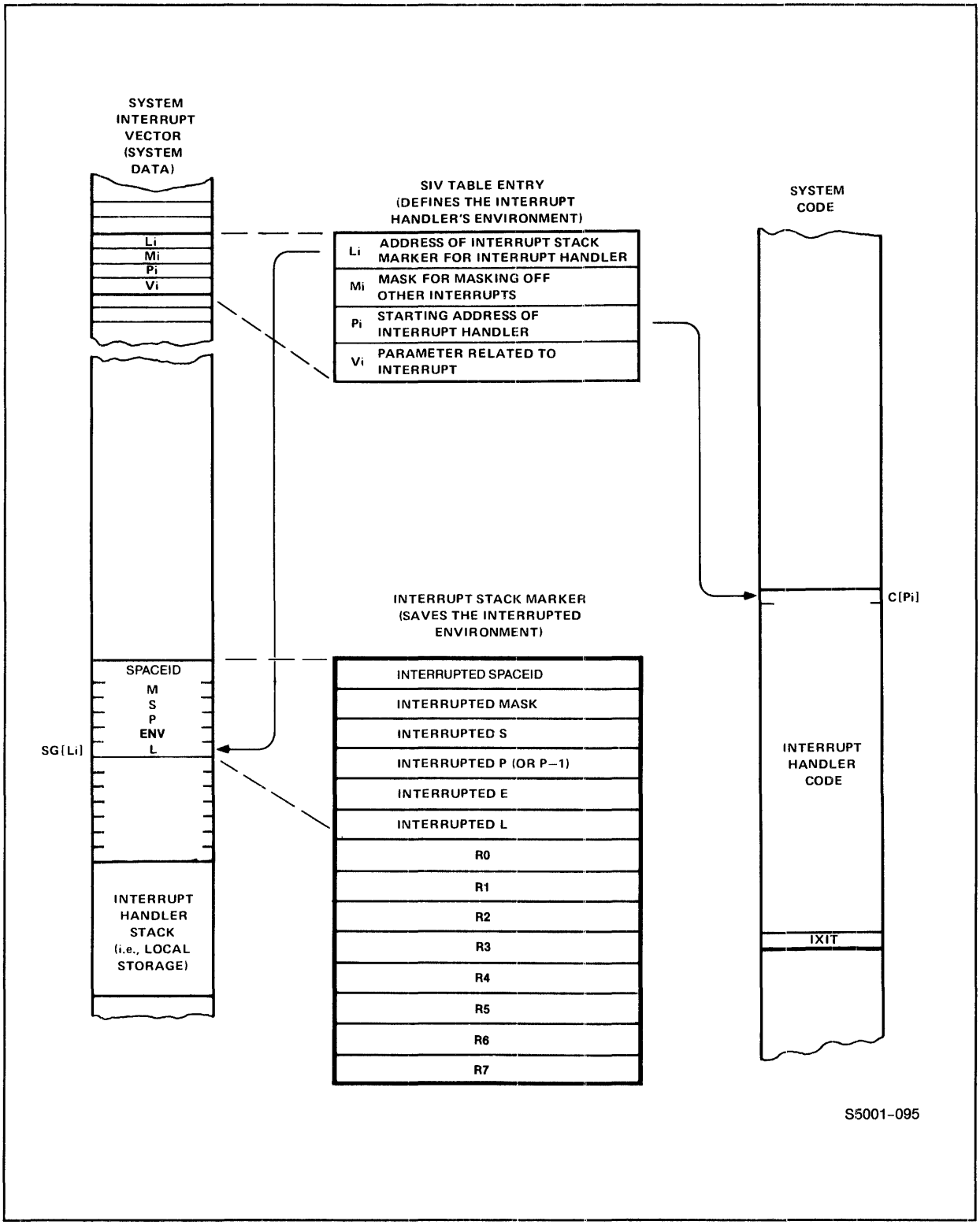


Figure 6-4. SIV Entry and Interrupt Stack Marker

- Mi: This is a mask value for masking off unwanted interrupts while an interrupt handler executes. The MASKi value in the SIV entry is ANDed with the current MASK register setting to derive a new setting. This permits nesting of interrupts of different types.

- Pi: This is the system code address of the interrupt handler's entry point.

- Vi: This is a location where an interrupt-related parameter may be returned by processor firmware.

INTERRUPT STACK MARKER

When an interrupt occurs, the interrupted environment is saved in an interrupt stack marker. The interrupt stack marker is placed at Li[-5:0] in the interrupt handler's stack; see Figure 6-4. The interrupt stack marker contains the following register values as they existed at the time of the interrupt:

```
Li[-5] = space ID, space identification of interrupted code
Li[-4] = M, the MASK register setting
Li[-3] = S, the S register setting
Li[-2] = P, the P Register setting
Li[-1] = ENV, the ENV register setting
Li[0]  = L, the L register setting
```

The format of the space ID is the same as is stored by a procedure call, described earlier in Section 4 (see Figure 4-24); that is, LS is in bit 4, CS is in bit 7, and the space ID index is in bits 11:15. Unlike the case of a procedure call, however, an interrupt saves the contents of the hardware ENV register intact and complete in Li[-1]; this is because the current CC and RP values must be restored on return from the interrupt.

In addition to the stack marker, each time an interrupt occurs the current contents of the Register Stack (R0 through R7) are saved in the first eight locations of local storage (i.e., sysstack[Li+1] through sysstack[Li+8]).

INTERRUPT SEQUENCE

An interrupt (i is the interrupt number) is defined as:

```
if INTA.<i> land MASK.<i> then      ! an interrupt occurred
  begin
    Vi := interrupt parameter;      ! if any
    sysstack[Li-5] := space ID;     !
    sysstack[Li-4] := MASK;         !
    sysstack[Li-3] := S;            ! interrupt stack marker
    sysstack[Li-2] := P;            !
    sysstack[Li-1] := ENV;          !
    sysstack[Li]   := L;            !
    sysstack[Li+1] := R0;           !
       thru                         ! saved Register Stack
    sysstack[Li+8] := R7;           !
    R7   := 2nd interrupt parameter;! if any; otherwise
                                    !  undefined
    ENV  := %3447;                  ! if NonStop II processor
                                    !  PRIV, DS, CS, V, RP = 7
    ENV  := %3507;                  ! if NonStop TXP processor
                                    !  PRIV, DS, CS, K, RP = 7
    L    := Li;
    S    := L + 8;
    P    := Pi;
    MASK := MASK LAND Mi;
  end;
```

An example is discussed in the following paragraphs, with reference to Figures 6-5 and 6-6. (The first 10 steps are shown in Figure 6-5.)

1. An interrupt condition occurs (in this example, a device is requesting standard I/O servicing).

   ```
   INTA.<14> := 1;
   ```

2. The current instruction completes executing and, since MASK.<14> is equal to 1, an interrupt occurs.

   ```
   if INTA land MASK then     ! interrupt.
     begin
   ```

3. There is no interrupt parameter for a standard I/O interrupt.

4a. The interrupted environment (including the current space ID, MASK and S register settings) is saved in the area pointed to by Li in the SIV entry for the standard I/O interrupt. The space ID is built by the interrupt microcode.

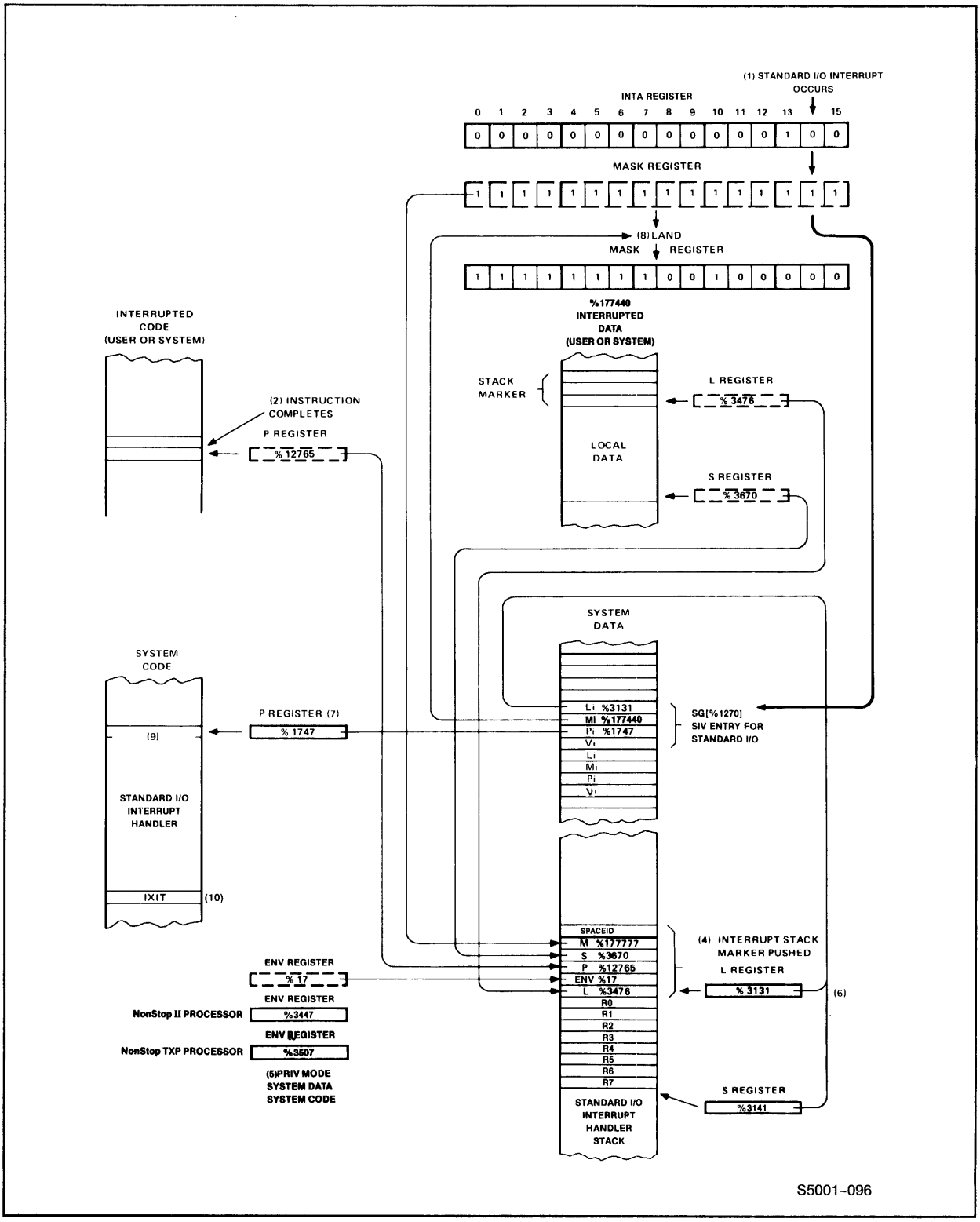


Figure 6-5. Interrupt Sequence

```
sysstack[Li-5] := space ID !
sysstack[Li-4] := MASK;    !
sysstack[Li-3] := S;       !
sysstack[Li-2] := P;       ! interrupt stack marker
sysstack[Li-1] := ENV;     !
sysstack[Li]   := L;       !
sysstack[Li+1] := R0       !
    thru                   ! saved Register Stack
sysstack[Li+8] := R7       !
```

4b. Register stack R7 receives the second interrupt parameter, if any; otherwise, R7's contents are undefined.

5. The PRIV (privileged mode), DS (data space), and CS (code space) bits in the ENV register are set. This defines the interrupt handler executing environment.

```
   ENV := %3447;        ! if NonStop II processor
or ENV := %3507;        ! if NonStop TXP processor
```

6. The L and S registers are set with the address of the interrupt handler's local data area. This is the value Li in the SIV entry for the standard I/O interrupt.

```
L := Li;
S := L + 8;
```

7. The P Register is set with the address of the first instruction in the standard I/O interrupt handler. This is the value Pi in the SIV entry for standard I/O.

```
P := Pi;
```

8. The Mi value in the SIV entry is ANDed with the current MASK register setting to derive a new MASK register setting.

```
MASK := MASK land Mi;
```

9. The first instruction of the standard I/O interrupt handler executes.

10. The interrupt handler runs to completion, unless the interrupt handler's mask allows interrupts or purposely unmasks any or all interrupts and corresponding interrupts do occur. Finally, an IXIT instruction is executed to return to the interrupted process.

11. The IXIT instruction (see Figure 6-6) restores the interrupted environment saved in the interrupt stack marker (at L[-5:0]); that is, the MASK, S, P, ENV, and L registers

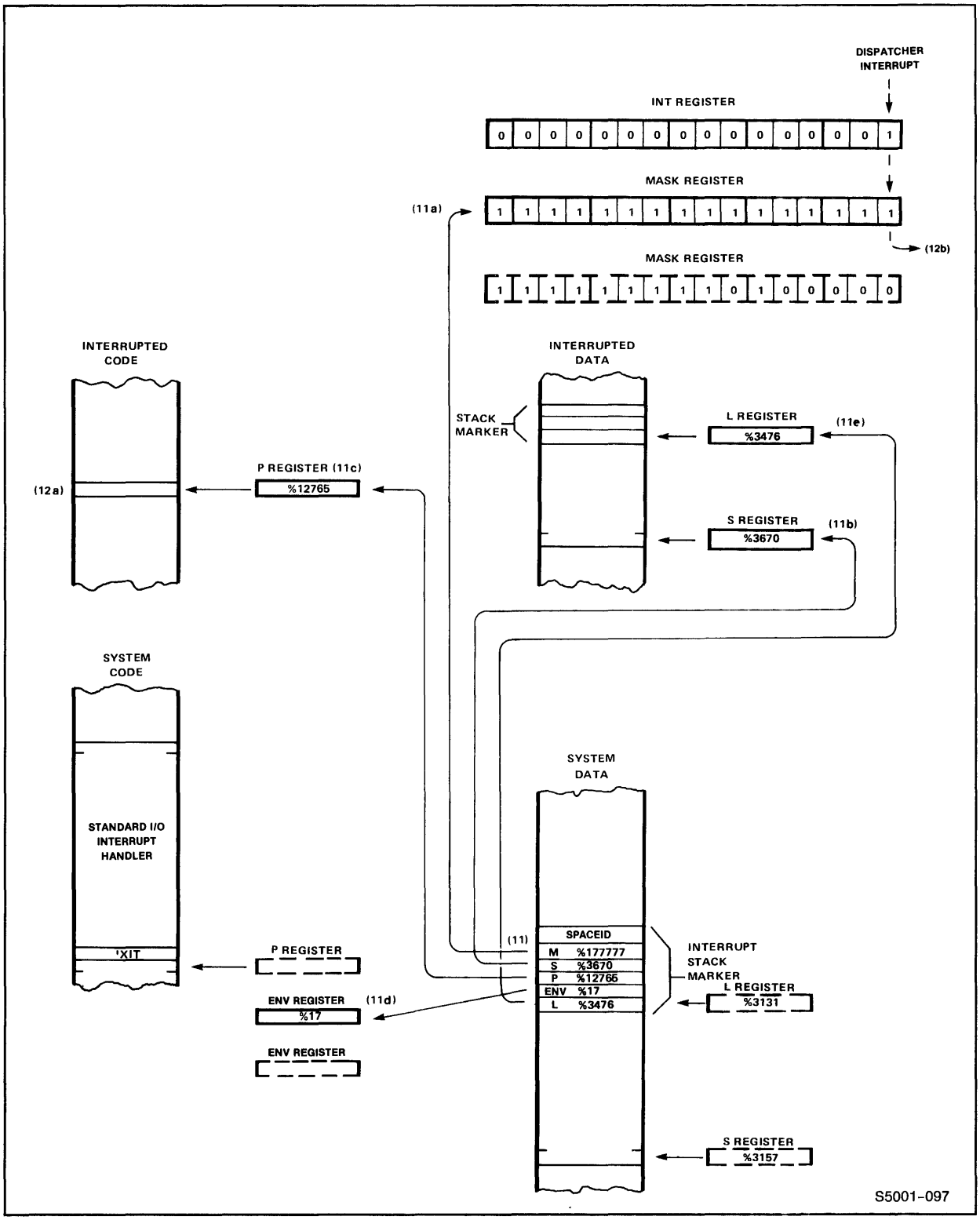


Figure 6-6. IXIT Sequence

are returned to their preinterrupt values, and the current space ID is restored.

```
MASK    := sysstack [L-4];     !  (a)
S       := sysstack [L-3];     !  (b)
P       := sysstack [L-2];     !  (c)
ENV     := sysstack [L-1];     !  (d)
L       := sysstack [L];       !  (e)
```

Also the Register Stack (values saved in L+1 through L+8) is returned to its pre-interrupt condition. If the segment being returned to is not currently mapped, the IXIT instruction automatically executes a MAPS (Map Segment) instruction, using the space ID information in L-5, prior to restoring the registers.

12a. If no interrupt is pending when the IXIT instruction completes, process execution resumes at the point of interruption.

12b. If another interrupt is pending, the interrupt sequence is repeated from step 1, using the appropriate SIV entry to set up the interrupt handler's environment.

INTERRUPT TYPES

The following paragraphs describe each of the interrupt types.

Special Channel Error (0). This interrupt occurs when the I/O channel detects types of errors that require software servicing. The error number is placed in the parameter word. Certain errors have a second error word giving the subchannel address and command, which is found in R7 on entry to the interrupt handler.

Uncorrectable Memory Error (1). This interrupt occurs when a memory word is accessed by the IPU and contains an error which cannot be corrected. The parameter contains the logical address of the page at fault and the six syndrome bits generated by the error correction circuitry. These syndrome bits provide information for Tandem service personnel.

For a NonStop II processor, the format of the parameter word is:

```
V1.<0:5>     =  logical page
V1.<6:11>    =  syndrome
V1.<12:15>   =  map number (SAS)
```

The contents of the data word that was in error are found in R7 on entry to the interrupt handler.

For a NonStop TXP processor, the parameter contains the MSTATUS word:

```
V1.<0:15>    =  MSTATUS word
```

The number of the physical page that contains the word in error is found in R7 in entry to the interrupt handler.

Memory Access Breakpoing (2). This interrupt occurs when the memory breakpoint has been armed by the SMBP instruction and the breakpoint memory address has been accessed in the desired manner. There is no parameter.

If a data page fault interrupt is pending, the processor clears the memory access breakpoint and processes the page fault. Any pending code page fault is cleared if the breakpoint is taken. No interrupt occurs if the breakpoint was armed by the Operations and Service Processor (OSP); in this case, the processor performs a system freeze and enters the idle loop.

Instruction Failure (3). This interrupt occurs when an unimplemented instruction is executed, or when execution of a privileged instruction is attempted by a program which is not in privileged mode, or when an abnormal condition is detected during the execution of certain instructions. The parameter for this trap is the current instruction.

Page Fault (4). This interrupt occurs when an attempt is made to access an absent memory page (i.e., its page table entry "absent" bit is set to 1). The parameter word is the high-order word of the absolute extended address of the absent page. R7 contains the low-order word of this address.

OSP I/O Completion (7). The I/O completion interrupt for the Operations and Service Processor occurs when either a read or a write operation to the OSP completes. The parameter word indicates the status, as follows:

0	normal read completion
1	normal write completion
%177777	character overrun detected on a read
%177776	write interrupt with negative byte count
%177775	read interrupt with zero or negative byte count

Power Fail (8). This interrupt occurs when a processor module power failure is detected. A minimum of five milliseconds is available for processing after this interrupt occurs before power is lost. There is no parameter.

Correctable Memory Error (9). This interrupt occurs when a memory error occurred and can be corrected. The parameter word is of the same form as that for an uncorrectable memory error.

The NonStop II processor is able to rewrite the page in place because its page table entry remains in a map register for the duration of the CMRW.

In a NonStop TXP processor, the CMRW instruction cannot tolerate a page table cache miss. Thus, the processor temporarily maps the errant page somewhere in segments 0-15 while it is being rewritten.

High-Priority I/O Completion (10). This interrupt occurs when a device that is connected to the high-priority interrupt poll line requires servicing. There is no parameter.

Interprocessor Bus Receive Completion (11). This interrupt occurs when a transmission is received on either the X-bus or the Y-bus. The parameter word is of the following form:

V11.<0> = bus flag
- 0 received on X-bus
- 1 received on Y-bus

V11.<1:7> = status
- 0 normal completion
- 1 unexpected packet
- 2 checksum error
- 3 misrouted packet or sending cluster/CPU unknown

4 unsequenced packet
5 sequence error
6 illegal extended buffer address

V11.<8:15> = cluster/processor number of sending processor

In addition, R7 contains the checksum+1 computed by the microcode when a checksum error is detected.

Time List (13). Every 10 milliseconds the microcode detects an interval clock micro-interrupt, updates the quadword clock at SG[103], and decrements the wait time of the element at the head of the Time List. If it has gone to zero, control passes to the time list interrupt handler; otherwise, no action is taken. There is no parameter.

Standard I/O Completion (14). This interrupt occurs when a device that is connected to the standard interrupt poll line requires servicing. There is no parameter.

Dispatcher (15). This interrupt occurs when a DISP or SNDQ instruction is executed, when a process-time timeout occurs, or when a PSEM or VSEM instruction is executed that requires operating system aid. Bit 15 of the parameter word is set on a DISP, bit 14 is set on a SNDQ, bits 13 and 15 are set on a PSEM when the semaphore cannot be obtained, and bit 12 is set when a VSEM instruction must release a blocked process. No part of the parameter word is ever cleared by the processor. If a Dispatcher interrupt is pending but the contents of the parameter word are zero, the interrupt is cleared.

Power On (16). This interrupt occurs when power is applied following a power failure when memory is in a valid state and the maps (NonStop II processor) or "dedicated half" of PCACHE (NonStop TXP processor) have been successfully loaded with no uncorrectable memory errors. The contents of Loadable Control Store are invalid. There is no parameter for this interrupt.

Stack Overflow (17). This interrupt occurs when S exceeds 32,767 (i.e., the limit of the memory stack) following the execution of any instruction that can change the S register setting--SETS, PCAL, XCAL, ADDS, BSUB, or PUSH. There is no parameter.

Arithmetic Overflow (18). This interrupt occurs when the T (trap enable) and V (arithmetic overflow) bits in the ENV register are simultaneously set to 1. There is no parameter.

Instruction Breakpoint (19). This interrupt occurs when a BPT instruction is executed, or when an EXIT or DXIT instruction is executed with ENV.<1> set to 1 in the stack marker. The parameter is the instruction which caused the interrupt.

XRAY Sampler (20). This interrupt, which only exists in the NonStop TXP processor, occurs when the sampler interval timer reaches zero. The sampler interval timer is a pseudo-random timer maintained by the DDT (the DDTX instruction enables and disables the timer). This interrupt is enabled only if XRAY sampling has been requested. There is no parameter word for this interrupt.

REENABLING INTERRUPTS

When an interrupt occurs, further interrupts of the same type are disabled while the current environment is being saved and the interrupt handler environment established. Interrupts of that type are automatically reenabled at the time of entry into the interrupt handler; however, interrupts masked by the setting of the Mi location in the SIV entry will still be prevented from occurring until the interrupt handler has completed. Mi must therefore be set to mask all unwanted interrupts. Note that this requires that Mi bits 11 and 12 both be zero when executing the interprocessor bus receive interrupt handler, to prevent an interrupt due to inbound traffic on the other bus.

SECTION 7

INTERPROCESSOR BUSES AND INPUT-OUTPUT CHANNEL

INTERPROCESSOR BUSES

A NonStop computer system has two interprocessor buses, designated the X-bus and the Y-bus. Each processor module in the system is connected to both buses and is capable of communicating with any processor module (including itself) over either bus. See Figure 7-1.

With any given interprocessor bus transfer, one processor module is the source (and initiator), and the other is the destination (and receiver). Before a processor module can receive data over

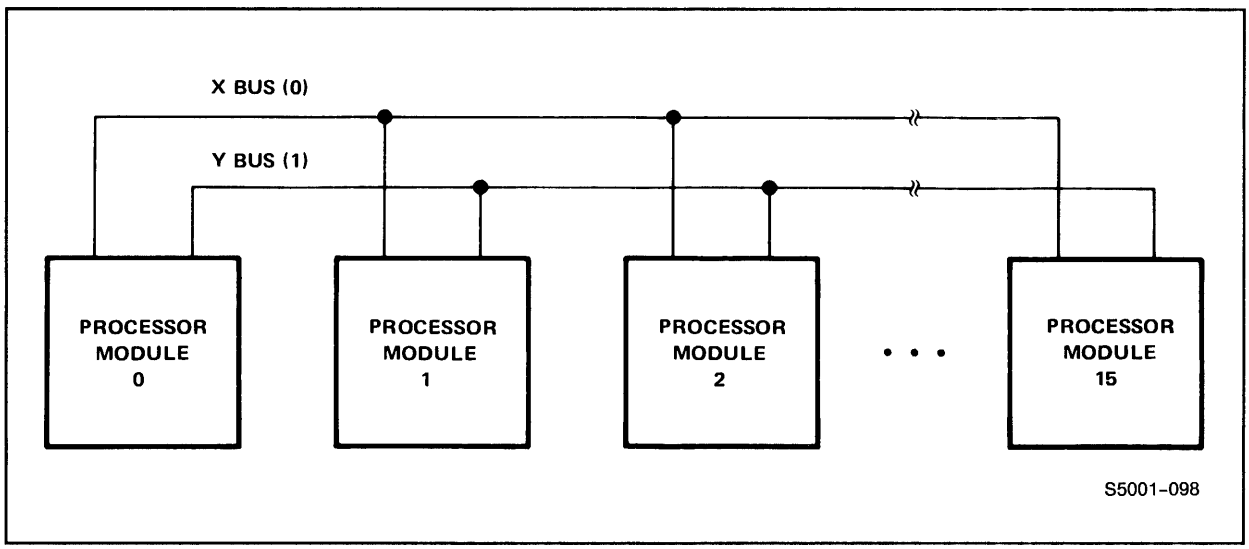


Figure 7-1. Processor Module Addressing

an interprocessor bus, the operating system first configures an entry in a table known as the Bus Receive Table (BRT). Each BRT entry contains, among other things, the address where the incoming data is to be stored and the number of bytes expected.

The FOX network is a fiber optic extension to the X- and Y-buses of the interprocessor bus. A FOX network establishes a high-speed communication link for a ring of systems composed of NonStop II and/or NonStop TXP processors. A ring can contain up to fourteen systems; each system, also known as a cluster, can contain up to sixteen processors.

The FOX network uses pass-through routing. Systems need not be connected directly to one another to exchange data; messages can be passed through intermediate systems, allowing the fiber optic links in the FOX network to connect the systems in a ring configuration rather than a star (each system directly connected to each other system).

To transfer data over a bus (see Figure 7-2), a SEND instruction is executed in the source processor module. The SEND instruction specifies the bus to be used for the transfer, the destination processor module, the number of bytes to be sent, the source location in memory of the data to be sent, the sender's processor number, a timeout value, and a sequence number.

While the source processor module is executing the SEND instruction and sending data over the bus, the firmware in the

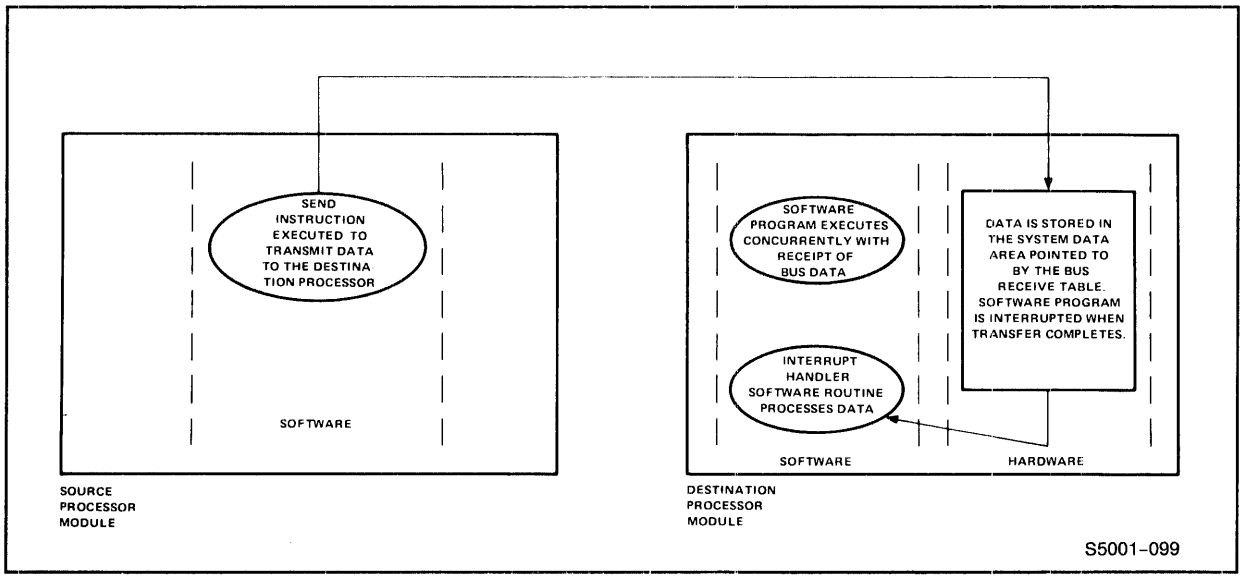


Figure 7-2. Simplified Bus Transfer Sequence

destination processor module is storing the data away according to the appropriate BRT entry (this occurs concurrently with program execution). When the destination processor module receives the expected number of bytes (the bus transfer is complete), an interprocessor bus receive interrupt is posted.

Bus Receive Table and Intercluster Bus Receive Table

The Bus Receive Table (BRT) contains 16 four-word entries, which correspond to the 16 processor modules possible in a system. The table begins at location SG[%1400].

Each entry in the BRT (see format in Figure 7-3) contains the address in virtual memory where the incoming data is to be stored, a count of the number of bytes expected, and the expected sequence number. (Refer to Section 5 for a description of virtual memory addressing using absolute extended addresses.)

If a processor is to receive data over a designated bus, the corresponding bit in the interrupt MASK register must be equal to 1. These mask bits, when on, enable both the receipt of data and the interrupt itself. The bits are:

X-Bus Receive Enable = MASK.<11>
Y-Bus Receive Enable = MASK.<12>

If a processor is part of a FOX network, its system has a unique cluster number in the range of 1-14. This cluster number, available to the microcode, is stored in location %154 of the system data segment, with the format shown in Figure 7-3.

Each system also considers itself to have a cluster number of 0 which it uses for all transfers that are local to its own interprocessor buses. The BRT table that starts at location %1400 of system data is treated as the BRT for cluster 0.

Cluster 15 is reserved for special functions (e.g., messages that require special handling by the bus controller). There cannot be an actual cluster number 15.

Another table, BRTLONG, points to the BRT entries for clusters 1 through 14 in the FOX network. BRTLONG contains sixteen 4-word entries, one entry per cluster.

The BRTLONG entry for a given cluster is located at:

SG [%1600 + 4 * cluster no.]

The format is shown in Figure 7-3.

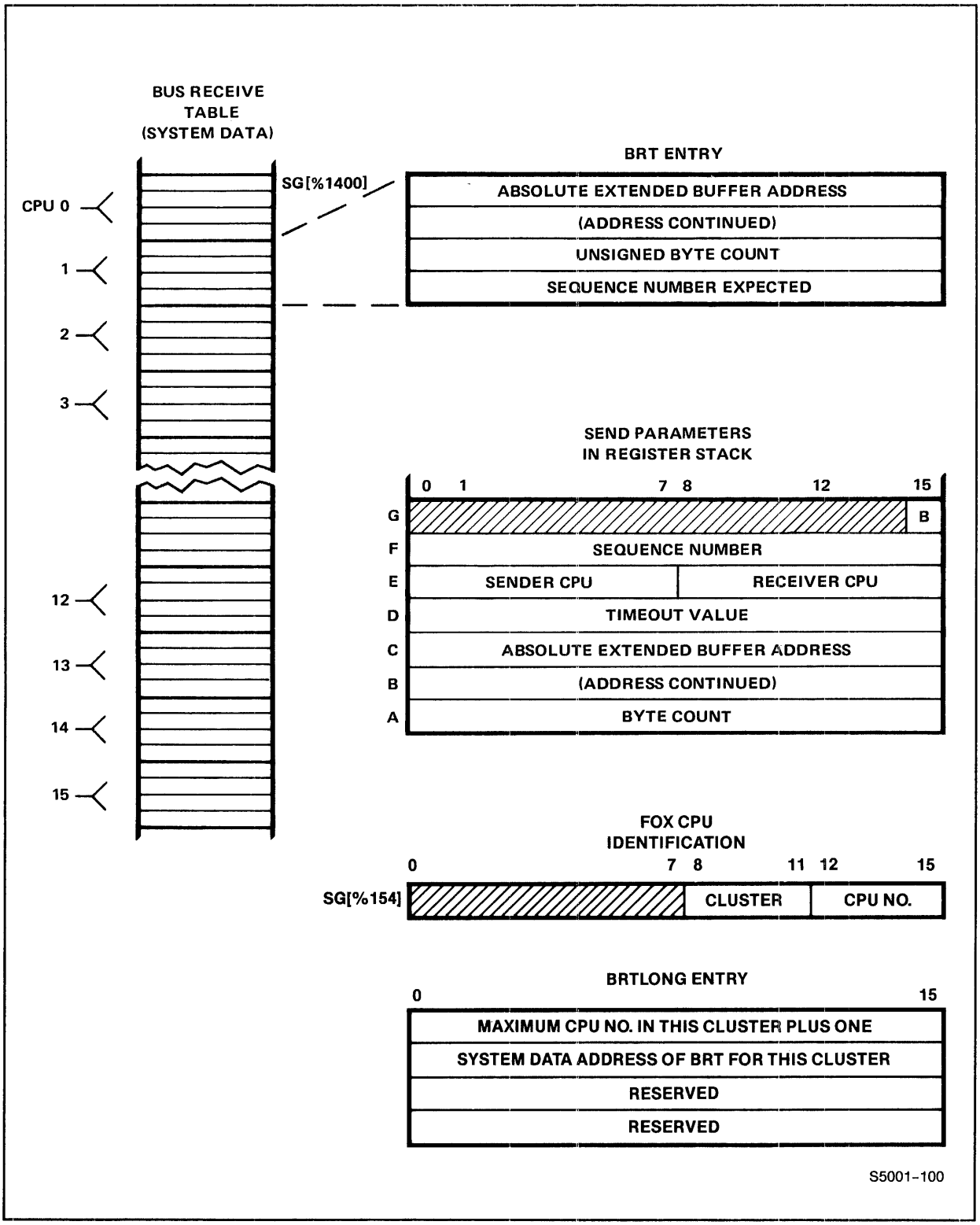


Figure 7-3. Formats Associated with Bus Transfers

SEND Instruction

The SEND instruction expects seven parameter words in the Register Stack. These are shown in Figure 7-3, and are described as follows.

- G.<15> specifies the bus (0 = X-bus, 1 = Y-bus) to be used.
- F.<0:15> is the sequence number to be sent.
- E.<0:7> specifies the sender processor module, and E.<8:15> specifies the receiver processor module.
- D.<0:15> is a value that is subtracted from 32,768 to derive the number of 0.8-microsecond units (NonStop II processor) or 0.833-microsecond units (NonStop TXP processor) allotted to completing a single packet (16-word) transfer. The timeout period is restarted for each packet transferred. (This parameter is normally zero when the operating system issues a SEND.)
- C.<0:15> and B.<0:15> form the absolute extended (byte) address of the buffer containing the data to be transferred.
- A.<0:15> is an unsigned count of the number of data bytes to be transferred.

Following execution of the SEND instruction, the Condition Code is set to either of two values:

CCL = Packet Timeout
CCE = Successful

Specifically, the SEND instruction executes as follows:

1. The IPU firmware checks whether the OUTQ is empty, since it must be empty when the send begins. If the OUTQ is not empty, the firmware checks for interrupts and services any that are pending. Then it checks for a timer overflow. If the timer did not overflow, it updates the timer and begins step 1 again. If a timer overflow occurred, indicating that the OUTQ did not become empty within the timeout period, a packet timeout occurs and the SEND is aborted. Timeout is defined as:

   0.8(32768 - D) microseconds (NonStop II processor)
   0.833(32768 - D) microseconds (NonStop TXP processor)

2. If data remains to be sent (i.e., count <> 0), it is placed in the OUTQ (bytes 4 through 29, or OUTQ[2:14]). If there are fewer than 26 bytes to be transferred, OUTQ[2:14] is

padded with zeros. The sequence number is placed in OUTQ[1] and the routing word in OUTQ[0]; an odd parity checksum is calculated and placed in OUTQ[15]. The packet is then sent, and the transfer address and count parameters are updated.

3. If no data remains to be sent, the SEND is flagged internally as "done" and the condition code is set to CCE to indicate a successful completion.

4. If a packet timeout occurs, the operation is also flagged internally as "done". However, the condition code is set to CCL to indicate a packet timeout.

5. The sequence repeats back to step 2 if the SEND is not "done".

Bus Transfer Sequence

As previously stated, there must be coordination between the source processor module and the destination module in regard to the number of bytes to be transferred. The operating system accomplishes this by preceding each transfer with a separate transfer (i.e., SEND) of a predetermined number of bytes of control information. In general, this control information tells the operating system in the destination module to expect a specified number of bytes over a specified bus. In the following example, illustrated in Figures 7-4a and b, assume that the initial transfer has taken place. The operating system in the destination module has configured the appropriate BRT entry for receiving 400 bytes.

1. A SEND instruction is executed in the source processor module (processor module 1). The SEND parameters specify:

   - X-Bus to Processor Module 3 (stack register G).
   - A sequence number (ignored in this example) (F).
   - Sender CPU 1 and receiving CPU 3 (E).
   - A packet timeout value of 0 (meaning that a timeout occurs if a single packet transfer takes longer than 26 milliseconds) (D).
   - A source buffer location address of 1466, which represents only the word and byte field values (11 bits of B) of the full 32-bit virtual memory address. (This is an absolute extended address. For simplicity, the other 21 bits of the address, representing the segment and page fields, are

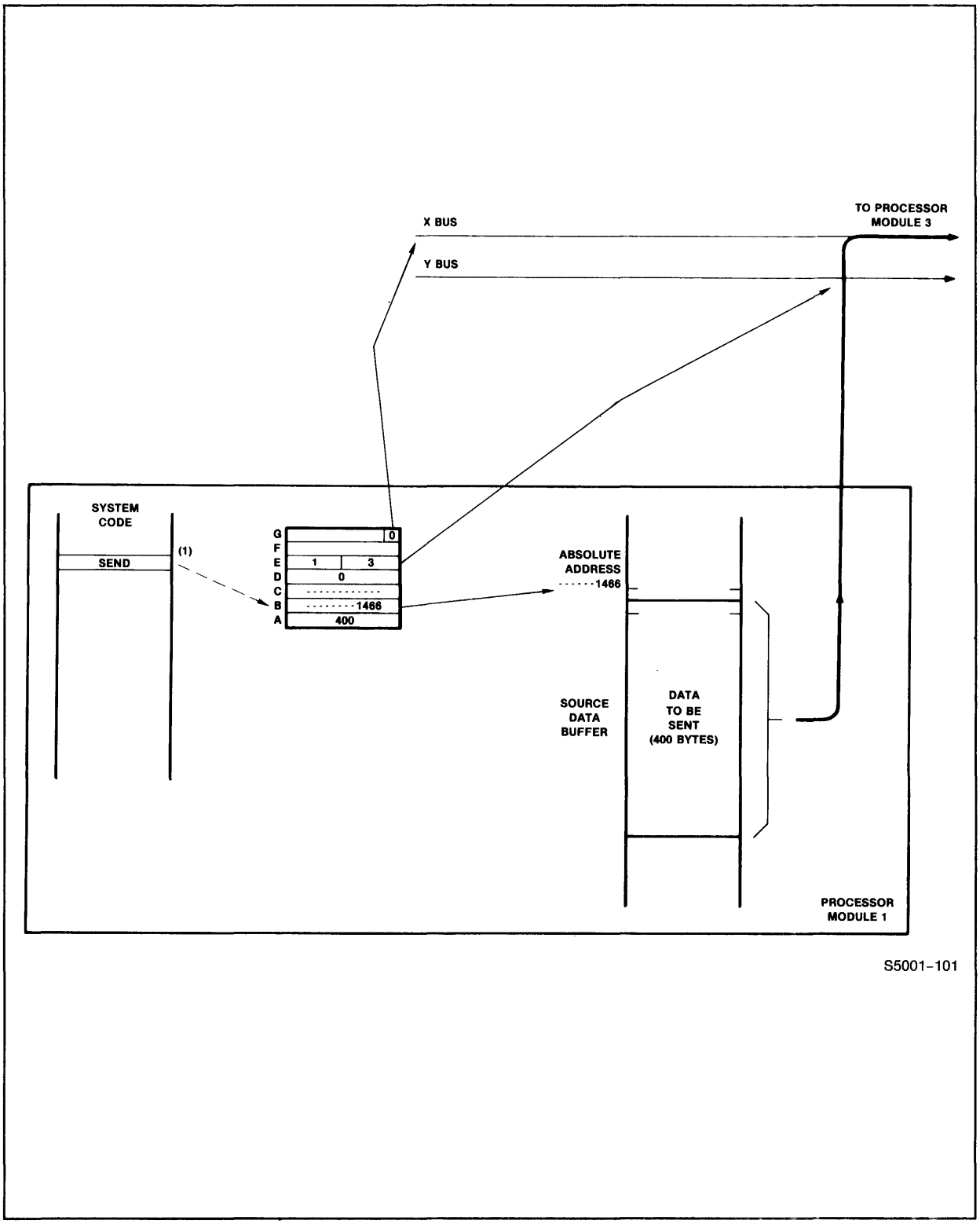


Figure 7-4a. Bus Transfer Sequence (Send)

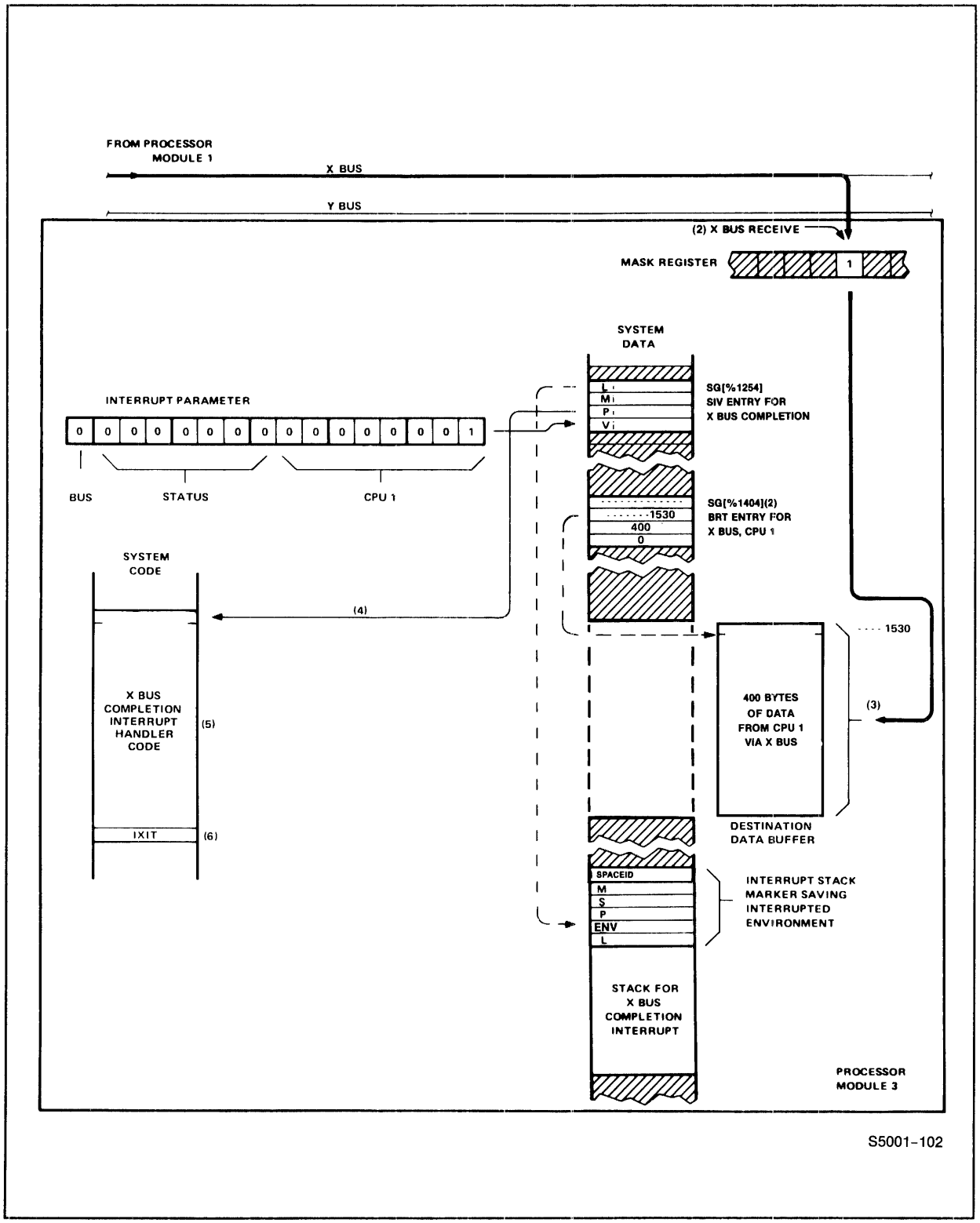


Figure 7-4b. Bus Transfer Sequence (Receive)

ignored throughout this example. Refer to the "Addressing" and "Memory Access" discussions for a description of virtual memory addressing using absolute extended addresses. Also note that since extended addresses are byte addresses, transfers on odd byte boundaries are permitted.)

- A count of 400 bytes to be transmitted (A).

The SEND instruction transmits the 400 bytes to processor module 3 via the X-bus, then completes. The parameters are deleted from the Register Stack and the condition code is set to CCE (indicating a successful operation).

2. Meanwhile, processor module 3, which has been previously readied for this transfer, has MASK.<11> set to 1 to enable receipt of data over the X-bus and has its BRT entry for processor module 1 configured as follows:

   - The transfer address where the incoming data is to be stored, starting at byte address 1530.
   - The count of the number of bytes expected, 400.
   - The initial sequence number.

3. The data, as received, is stored away as indicated by the BRT entry. As the data is stored, the transfer address is incremented accordingly and the count is decremented accordingly.

4. When the count in the BRT entry reaches zero, 400 bytes have been received. At this point an interrupt occurs through the SIV (System Interrupt Vector) for interprocessor bus completion. The parameter associated with this type of interrupt contains the processor module number of the source processor module, the bus flag (0 in this example), and the status (also 0 in this example).

5. The interrupt handler code for bus completion now executes. Because INT.<11> in the interrupt register is now set, further data transmissions to this processor module over the X-bus are rejected. Additionally, the Mi word in the SIV entry for bus completion masks off further interrupts in the MASK.<11:12> positions.

6. When the IXIT instruction executes, the previous MASK register setting is restored. Since the interrupt handler has already reset INT.<11>, processor module 3 is again enabled for receiving data over the X-bus.

Figure 7-5 shows the relationships of the transfer address, count, and sequence number in the BRT entry, and also the incoming data storage in the transfer location.

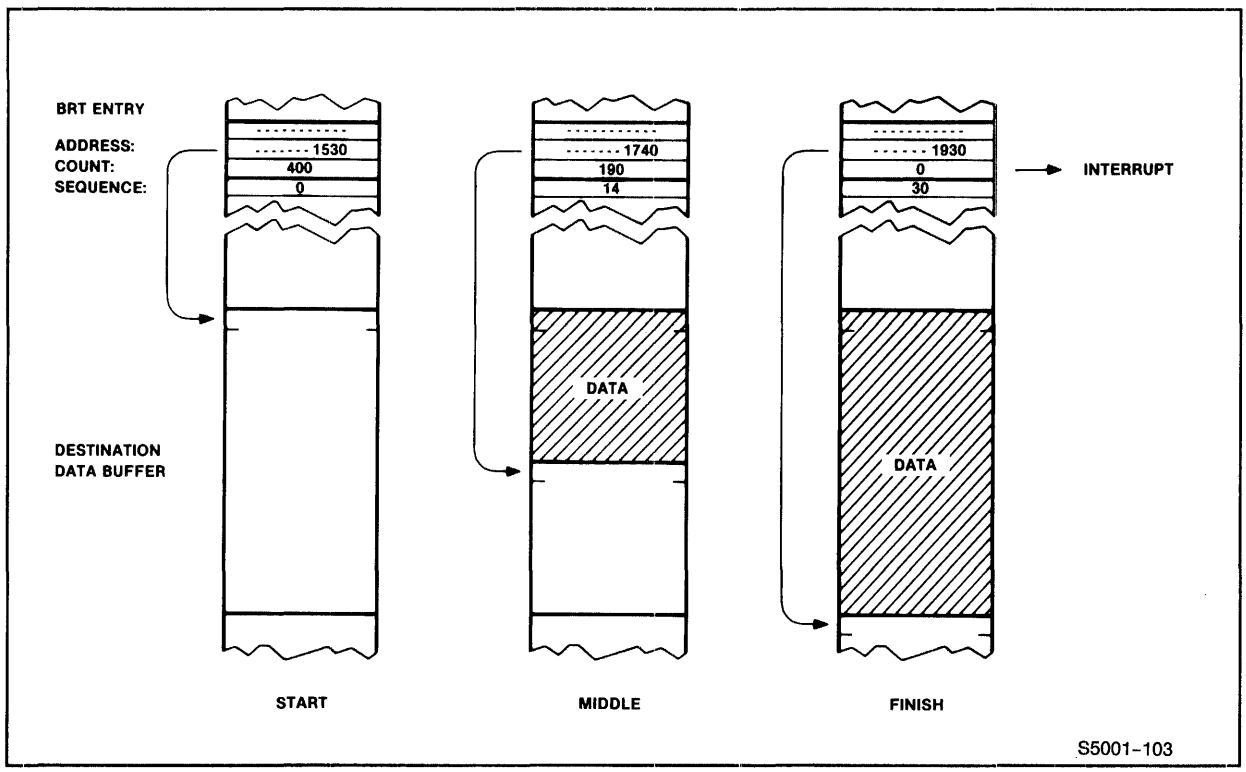


Figure 7-5. Incoming Data Storage

OUTQ, INQ, and Packets

The interprocessor buses are significantly faster than memory. Therefore each processor has a buffered interface to both buses; NonStop II processors have two 16-word output buffers (called OUTQ X and OUTQ Y), NonStop TXP processors have one 16-word output buffer (called OUTQ); both processor types have two 16-word input buffers (called INQ X and INQ Y). See Figures 7-6a and b.

Data is transmitted over a bus in the form of 16-word packets. The SEND instruction fills the output buffer with 26 data bytes (13 words), plus a one-word sequence number, one word for sender and receiver numbers, and a one-word odd-parity checksum. The instruction then signals the bus interface hardware that it has a

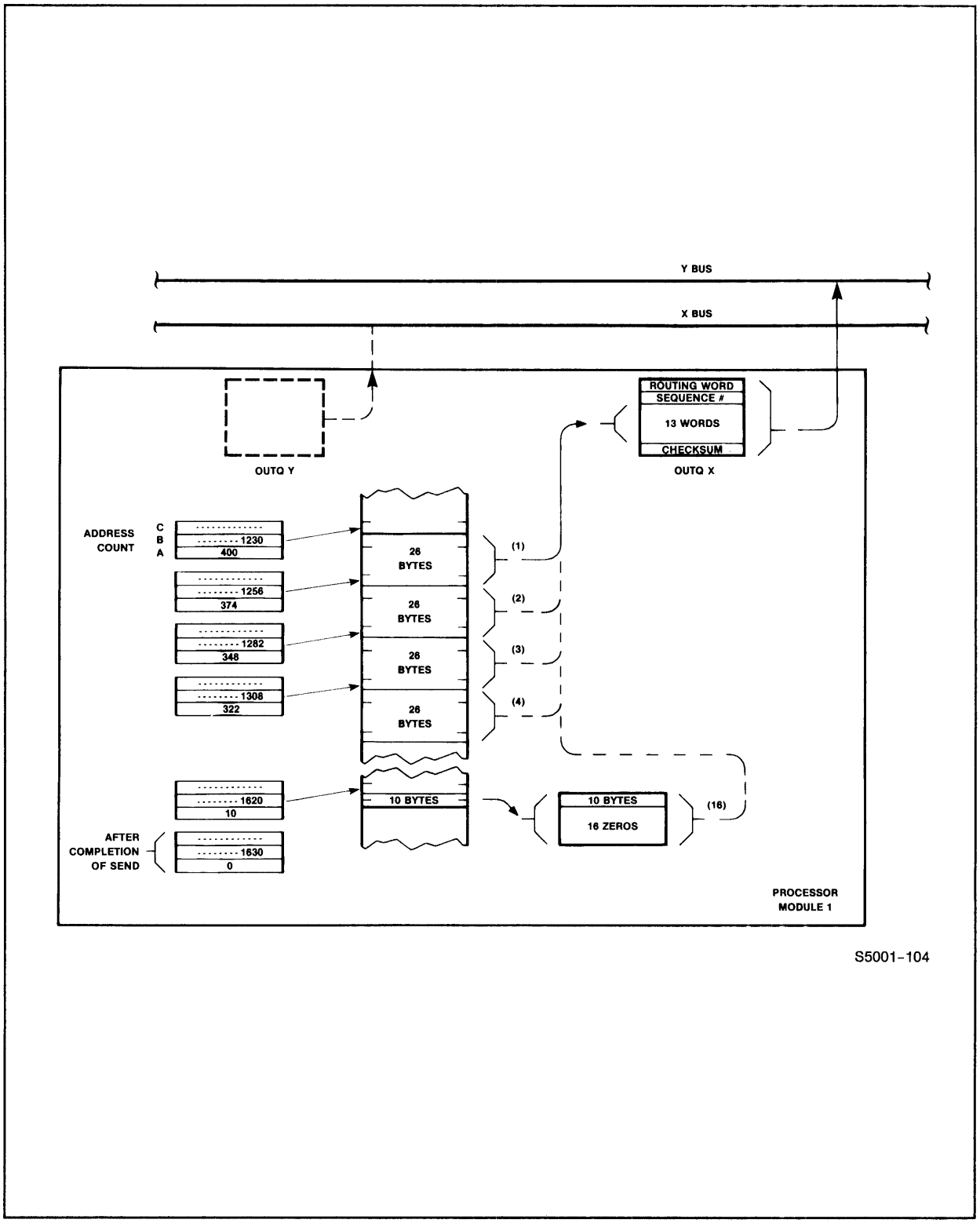


Figure 7-6a. Sending and Receiving Packets

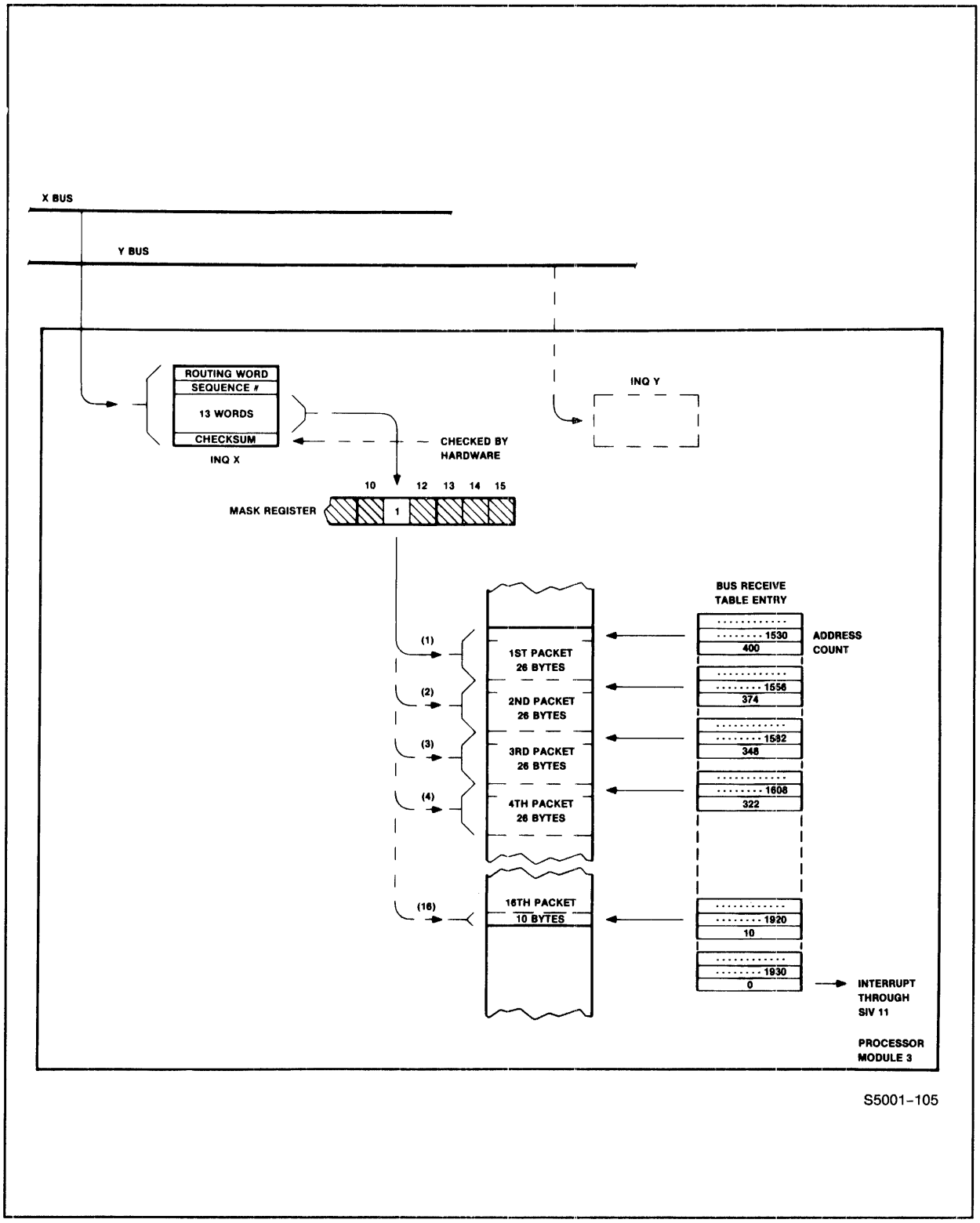


Figure 7-6b. Sending and Receiving Packets

packet ready for transmission. After the 16-word packet is transmitted, execution of the SEND instruction resumes at the point where it left off. If the last packet of the block contains less than 26 data bytes, the remaining data bytes are filled in with zeros. The SEND instruction terminates when the last packet is transmitted.

When either of the INQ X or INQ Y buffers in the destination processor module is filled and the corresponding MASK register bit is equal to 1, a microinterrupt occurs. The action taken by the processor module during the microinterrupt (which is transparent to the executing process and to the operating system) is:

- The count in the BRT entry is checked. If the count indicates that data is expected, 26 bytes (or less if the count is less) are read into memory at the location specified. The transfer address and count are then updated accordingly.

- The checksum of the packet is checked. If the checksum is valid and the count still exceeds zero, the INQ is marked empty (permitting further transmissions to take place) and the normal instruction execution sequence continues.

- If the count is now zero or if any transmission error is detected (checksum error, incorrect target, sequence error, etc.), the INT register bit associated with the bus used for the transmission is set to 1, and an interrupt occurs. In the case of a transmission error, the count word is not updated. When a normal receive completes, the count word will contain zero.

INT and MASK Registers

These registers have a direct bearing on the ability of a processor module to accept data over an interprocessor bus. As shown in Figure 7-7, data packets from the buses are accepted into INQ X or INQ Y whenever the data is sent to this module (provided that the INQ is empty). Once the data is accepted, the corresponding bit in the interrupt register (bit 11 and/or 12 of INTA) is then set. If the corresponding bit of the MASK register is also set (i.e., MASK and INTA bits ANDed together), a Bus Receive interrupt occurs that causes the IPU to transfer data to memory.

If a source processor module attempts a SEND to a processor module that is not enabled for receiving data (MASK bit inhibits destination processor from emptying its INQ), the source module receives a packet timeout indication.

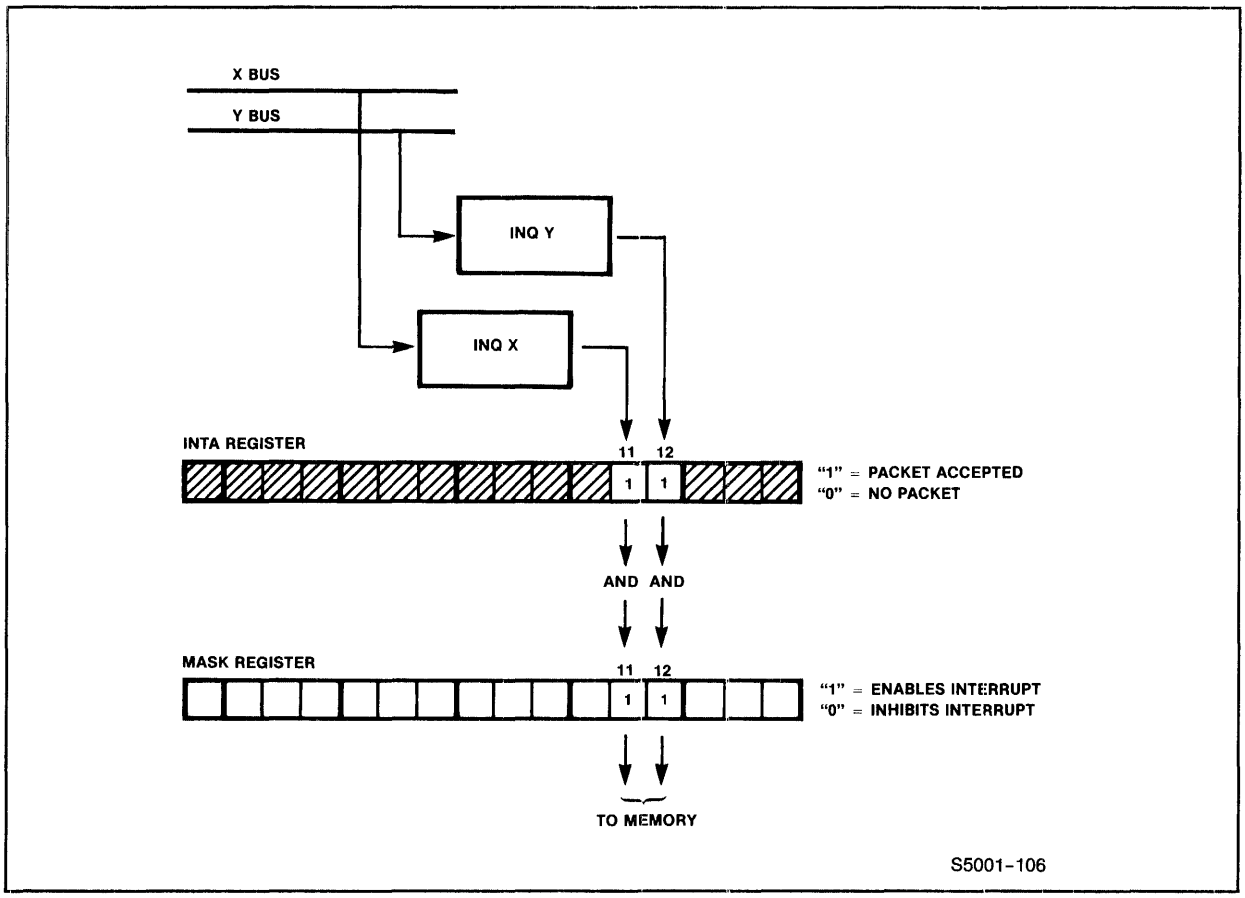


Figure 7-7. Bus Receive Enabling

INPUT-OUTPUT CHANNEL

Each processor module has a single block-multiplexed input-output channel through which all input-output takes place. Device-dependent I/O controllers are attached to the channel, and each controller may have one or more subchannels. A processor may address up to 256 subchannels. See Figure 7-8. Each controller is connected to two different processors, and the subchannel numbers that it responds to need not be the same on both processors. (Dual-port operation is considered later in this section.)

The first subchannel number for a given controller must be a multiple of 8, and the remaining subchannels follow in consecutive order.

The operating system performs input-output operations (see Figure 7-9) by first configuring an entry in a system table called the I/O Control Table (IOC). The IOC contains 256 entries, one for each subchannel that can possibly communicate over the I/O channel. Each entry contains the address of the data buffer and a count of the number of bytes to be transferred. Once the entry corresponding to the device is configured, an EIO (Execute I/O) instruction is executed to initiate the I/O transfer. When the transfer completes, an interrupt to an operating system interrupt handler takes place. In the interrupt handler, an IIO (Interrogate I/O) instruction or an HIIO (High-priority Interrogate I/O) instruction is executed to check the outcome of the operation.

I/O Control Table

The data to be transferred between memory and a specific unit is determined by an entry in the I/O Control Table (IOC). As illustrated earlier (Figure 5-21), this table occupies all of the second page of the system data segment. It contains a four-word entry for every possible subchannel which may be connected to a processor module. See Figure 7-10.

The first word of the the IOC entry specifies the base address of the I/O buffer in virtual memory. Bits 6 through 9 specify the absolute segment number (6:13), and bits 10 through 15 specify the starting logical page number within the segment. It is permissible for I/O buffers to cross address space boundaries.

The second word of the IOC entry specifies the number of bytes remaining to be transferred. This value is decremented after each word transfer.

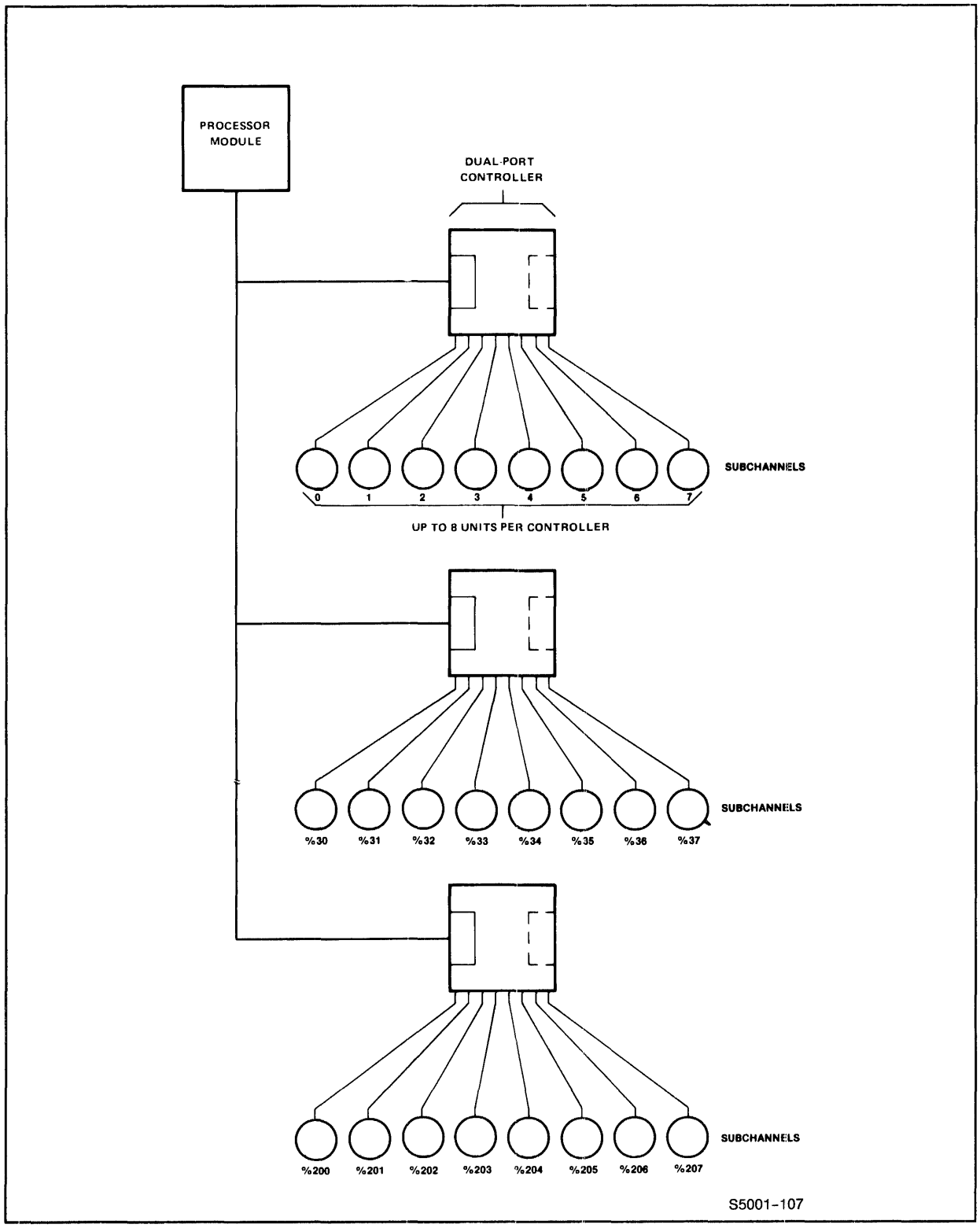


Figure 7-8. I/O Channel Addressing

The third word of the IOC entry specifies the current word in the buffer that needs to be transferred. Since the segment offset value given in bits 0 through 5 is relative to the page base value given in the first word of the entry, these two values are added together to derive the actual logical page in memory currently being accessed for word transfers. This value is incremented after each word transfer.

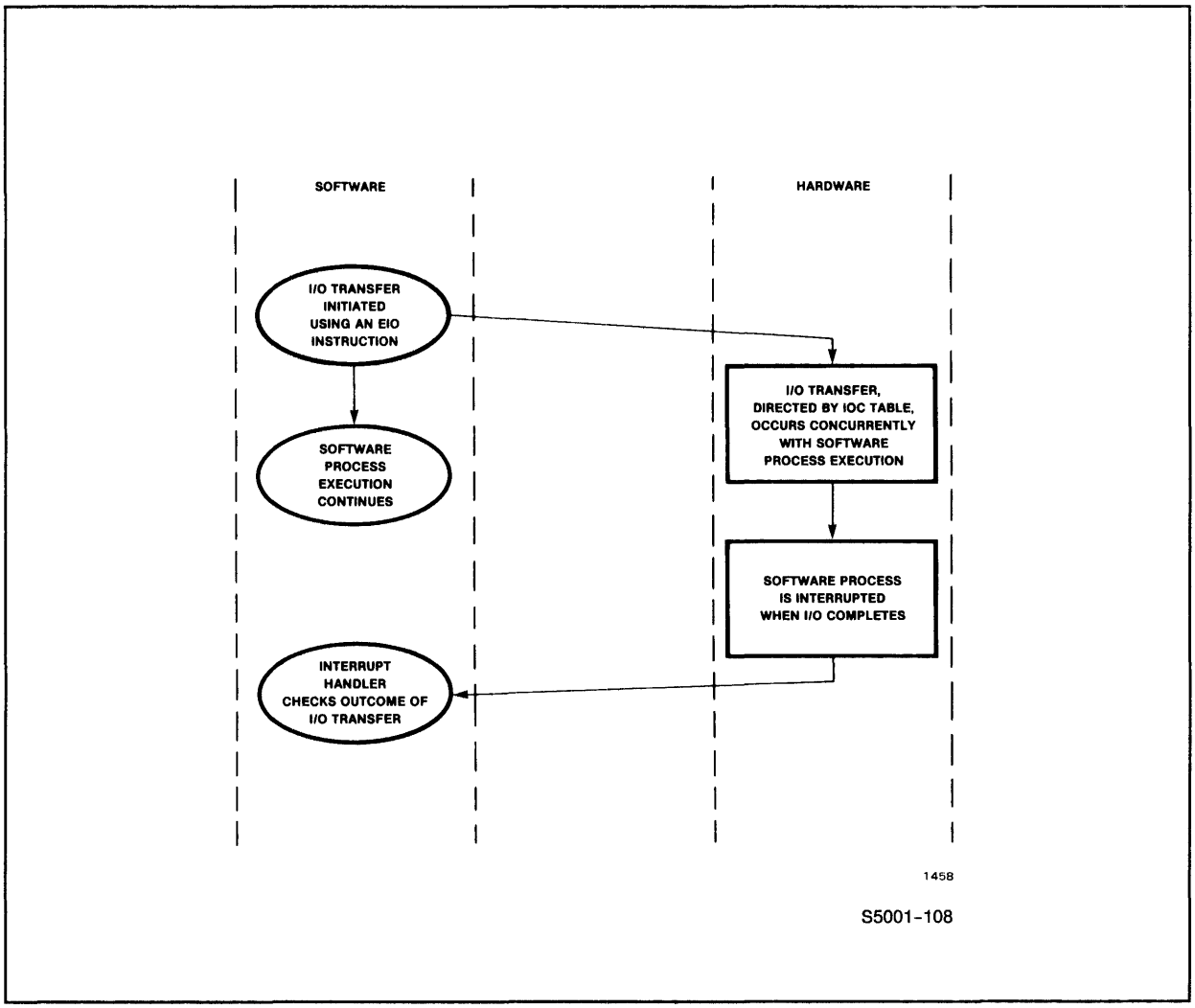


Figure 7-9. Simplified I/O Sequence

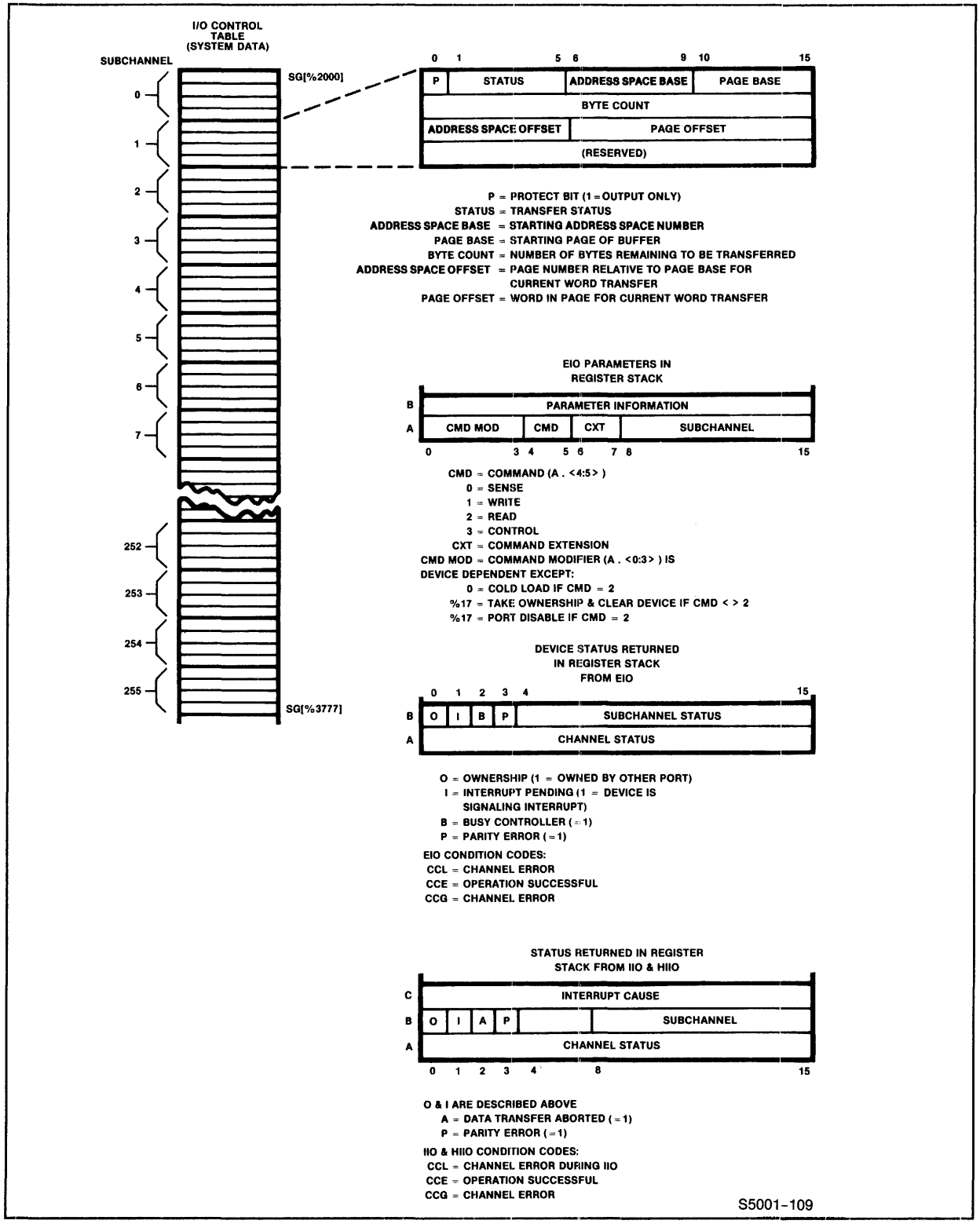


Figure 7-10. Formats Associated with Input-Output

To prevent erroneous data transfers, the operating system either sets the second word in the IOC entry to zero when transfers are not expected, or, if the last transfer was outbound, sets the protect bit. If a device attempts to transfer data when the byte count is zero, the I/O channel aborts the operation, causing an interrupt to occur. In such a case, the status returned by the device as a result of an IIO or HIIO reflects the error.

The NonStop TXP processor caches active IOC entries, and updates only the cache copy during the transfer. As a result, the operating system must copy each entry that will be used into the IOC cache before the EIO instruction is issued. Similarly, the operating system must copy each entry that it wishes to inspect, during or after the transfer, back to the corresponding IOC entry in memory. Two instructions in the NonStop TXP processor (supported as no-ops in the NonStop II processor) perform these functions:

- LIOC copies the specified subchannel's IOC entry from the memory-resident IOC Table to the IOC cache which is resident in scratchpad registers.
- SIOC copies the specified IOC entry from scratchpad back to the memory copy of the IOC Table.

A third instruction, XIOC, exchanges the specified subchannel's IOC entry with the IOC entry that is currently in the scratchpad--basically a combination of SIOC with LIOC. This instruction is fully supported on both processor types.

EIO Instruction

To perform an I/O operation, the IOC entry for the unit must first be correctly initialized. (In a NonStop TXP processor, the entry must be cached in scratchpad registers via the LIOC instruction.) An EIO instruction can then be executed, specifying the controller, unit, command, and other parameter information. These parameters are placed in B and A of the Register Stack. (See format in Figure 7-10.)

The parameters to the EIO instruction are described as follows:

- The parameter information word in B is a device-dependent parameter that is sent to the specified device.
- Command bits A.<0:5> specify the operation that the device is to perform. The CMD bits, A.<4:5>, specify the general type of command:

0 = sense
1 = write
2 = read
3 = control

The CMD MOD bits, A.<0:3>, modify the command, allowing up to 64 device-dependent commands.

Three configurations of these fields are reserved:

CMD	CMD MOD	Description
2	0	perform cold load
3	%16	disable port (kill)
3	%17	take ownership and clear device

- The CXT bits, A.<6:7>, are available as command extension bits, specific to each device that requires them.
- The subchannel field, A.<8:15>, specifies one of 256 subchannels.

The EIO instruction replaces the two parameter words by two words containing the device status and the channel status, and sets the Condition Code according to the outcome of the instruction. The Condition Code settings are as follows:

CCL: channel error (while executing EIO)
CCE: operation successful
CCG: channel, controller, or device error

The device status is of the form:

B.<0> = ownership
B.<1> = interrupt pending
B.<2> = busy
B.<3> = parity error
B.<4:15> = subchannel status

The status bits returned in B have the following meanings:

- O (ownership), B.<0>, is equal to 1 if the device is owned by the other port. No data is transferred.
- I (interrupt pending), B.<1>, is equal to 1 if the device is interrupting. No data is transferred.
- B (busy), B.<2>, indicates that the device is already executing an I/O transfer (this includes seeking on a disc or rewinding on a magnetic tape). No data is transferred because of this EIO.

- P (parity), B.<3>, indicates (if equal to 1) that a parity error occurred.

The channel status word returned in A can have the following values:

%000000	no error detected in the channel
%000100	device status <0:3> non-zero
%000200	channel detected a parity error on RIC
%000400	channel detected a parity error on RIST or RDST
%1XXXXX	channel status = IOBUS Control field

IIO and HIIO Instructions

Following the successful initiation of an I/O operation by an EIO instruction, an interrupt occurs when the operation completes. At this point, an IIO (Interrogate I/O) or HIIO (High-Priority Interrogate I/O) instruction must be executed to determine the cause of the interrupt. When the IIO or HIIO is executed, the highest-priority device with an interrupt pending returns its subchannel number and a three-word status pertaining to the interrupt.

The three status words returned to the Register Stack by the execution of an IIO or HIIO instruction are of the form:

C.<0:15>	=	interrupt cause
B.<0>	=	ownership
B.<1>	=	interrupt pending
B.<2>	=	aborted
B.<3>	=	parity error
B.<8:15>	=	subchannel number
A.<0:15>	=	channel status

The status bits have the following meanings:

- The interrupt cause field, C.<0:15>, is related to the particular subchannel that is interrupting.
- O (ownership), B.<0>, is equal to 1 if the controller is owned by the alternate port (see the description of "Dual-Port Controllers and Ownership" that follows).
- I (interrupt pending), B.<1>, is equal to 1 if the device has an interrupt pending. Normally this bit should not be set at this time; if it is set, some problem is indicated.
- A (aborted), B.<2>, is equal to 1 if the data transfer was aborted.

- P (parity error), B.<3>, is equal to 1 if a parity error was detected during the data transfer sequence.

- The subchannel field, B.<8:15>, is the controller and unit number associated with the interrupt.

- The channel status field, A.<0:15>, defines a possible channel error and may have the following values:

%000000	no error detected by the channel
%000100	device status bits <0:3> nonzero
%000200	channel detected a parity error on RIC (Read Interrupt Command)
%000400	channel detected a parity error on RIST (Read Interrupt Status) or RDST (Read Status)
%177777	instruction timed out waiting for the I/O channel to become available
%1-----	channel status = IOBUS Control Field

Following execution of an IIO or an HIIO instruction, the Condition Code is set as follows:

CCL:	channel error (while executing the instruction)
CCE:	operation successful
CCG:	channel, controller, or device error

Input-Output Sequence

A typical data transfer sequence over the input-output channel is depicted in Figure 7-11. The sequence is as follows:

1. Instructions in the I/O driver procedure are executed to configure the IOC entry for the subchannel through which the transfer is to take place. In this case, the IOC entry is at SG[%2030] for subchannel 6.

   For a NonStop TXP processor, the initialized IOC entry must be moved into the IOC cache by an LIOC instruction.

2. The EIO parameters are loaded onto the Register Stack.

3. An EIO instruction is executed. The parameter information is sent to subchannel 6.

4. To indicate its outcome, the EIO instruction returns two status words to the top of the Register Stack and sets the Condition Code. These are checked by subsequent instructions.

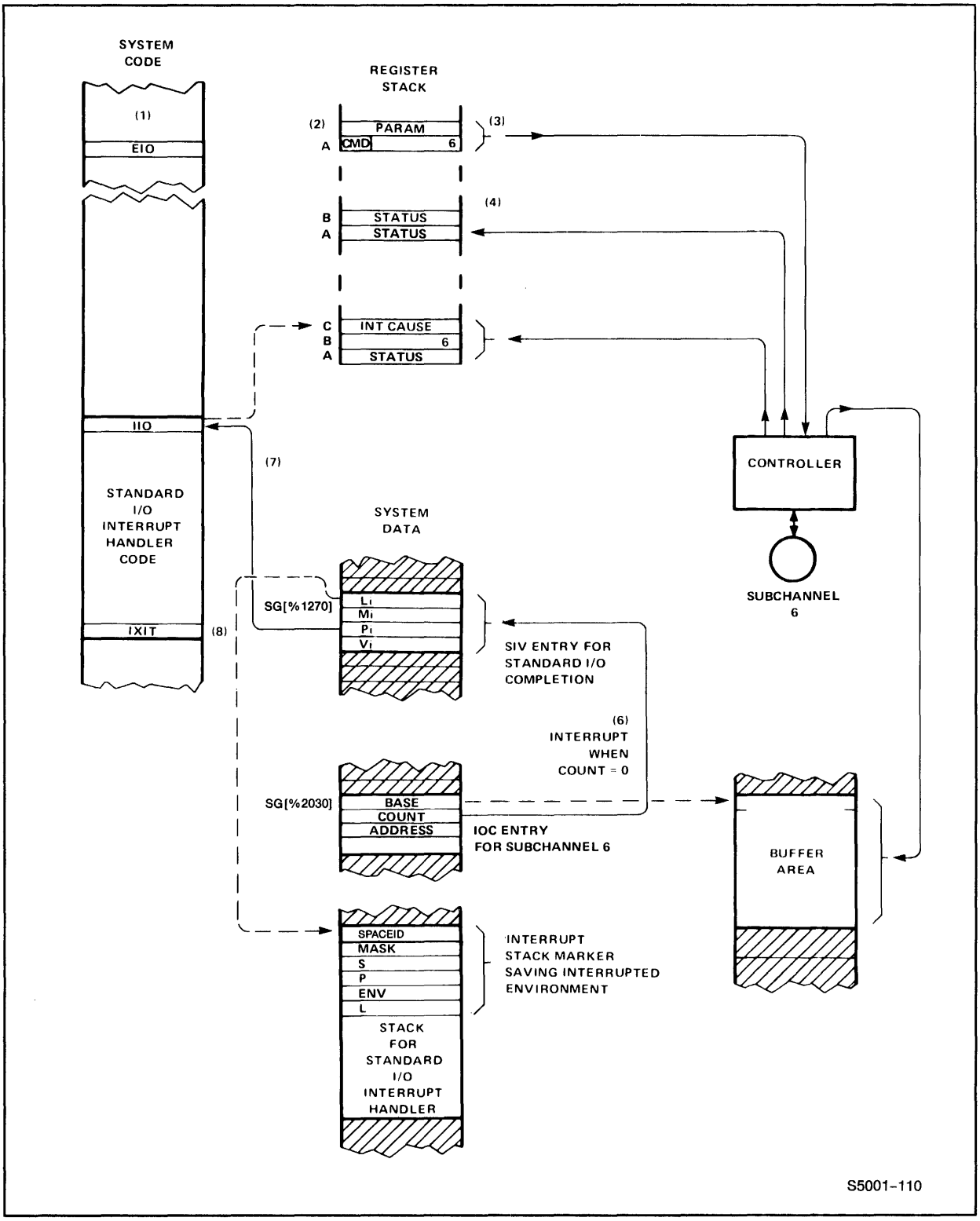


Figure 7-11. Input-Output Sequence

5. Meanwhile, the data transfer takes place. Data is transferred from subchannel 6 to the location in memory indicated by the IOC entry for that subchannel. As the data is transferred into memory, the transfer address and count word in the IOC are updated accordingly.

   For a NonStop TXP processor, the cached copy of the IOC entry is updated rather than the memory copy.

6. When the count word in the IOC reaches zero, indicating that the transfer is completed, the channel signals the controller. The controller stops transferring and signals the IPU with an interrupt. The INTA.<14> bit in the interrupt register is set to 1 to signal interrupt pending. If the corresponding bit in the MASK register is set, an interrupt through the SIV entry for standard I/O (at SG[%1270]) occurs. The Mi entry in the SIV causes any further standard I/O interrupts to be deferred while the I/O completion interrupt handler is active.

7. The interrupt handler executes an IIO instruction. Executing IIO signals the highest-priority interrupting controller to stop interrupting and returns three words of status information to the top of the Register Stack. (Controller priorities are set into the hardware at installation time, and may be adjusted by Tandem field service representatives as necessary for load balancing.) The status words contain the subchannel number of the interrupting device as well as interrupt cause and channel status information.

   For a NonStop TXP processor, the IOC entry must be retrieved from its scratchpad register copy and written back to the memory copy of the entry before its contents can be inspected. The SIOC instruction performs this function.

8. When the interrupt handler for standard I/O completes, an IXIT instruction is executed. IXIT restores the previous MASK register value (which allows any pending standard I/O interrupt to occur) and attempts to return control to the interrupted code. Typically the operating system intervenes at this point and the I/O process and, later, the user process are notified of the completion of the original I/O request.

Dual-Port Controllers and Ownership

Each controller in the NonStop II and NonStop TXP computer system is connected to the I/O channels of two processor modules. This provides redundant communication paths to I/O devices. As shown in Figure 7-12, this means that a single subchannel has entries in the IOCs of two processor modules. Note that the ports need not have the same subchannel address on both channels.

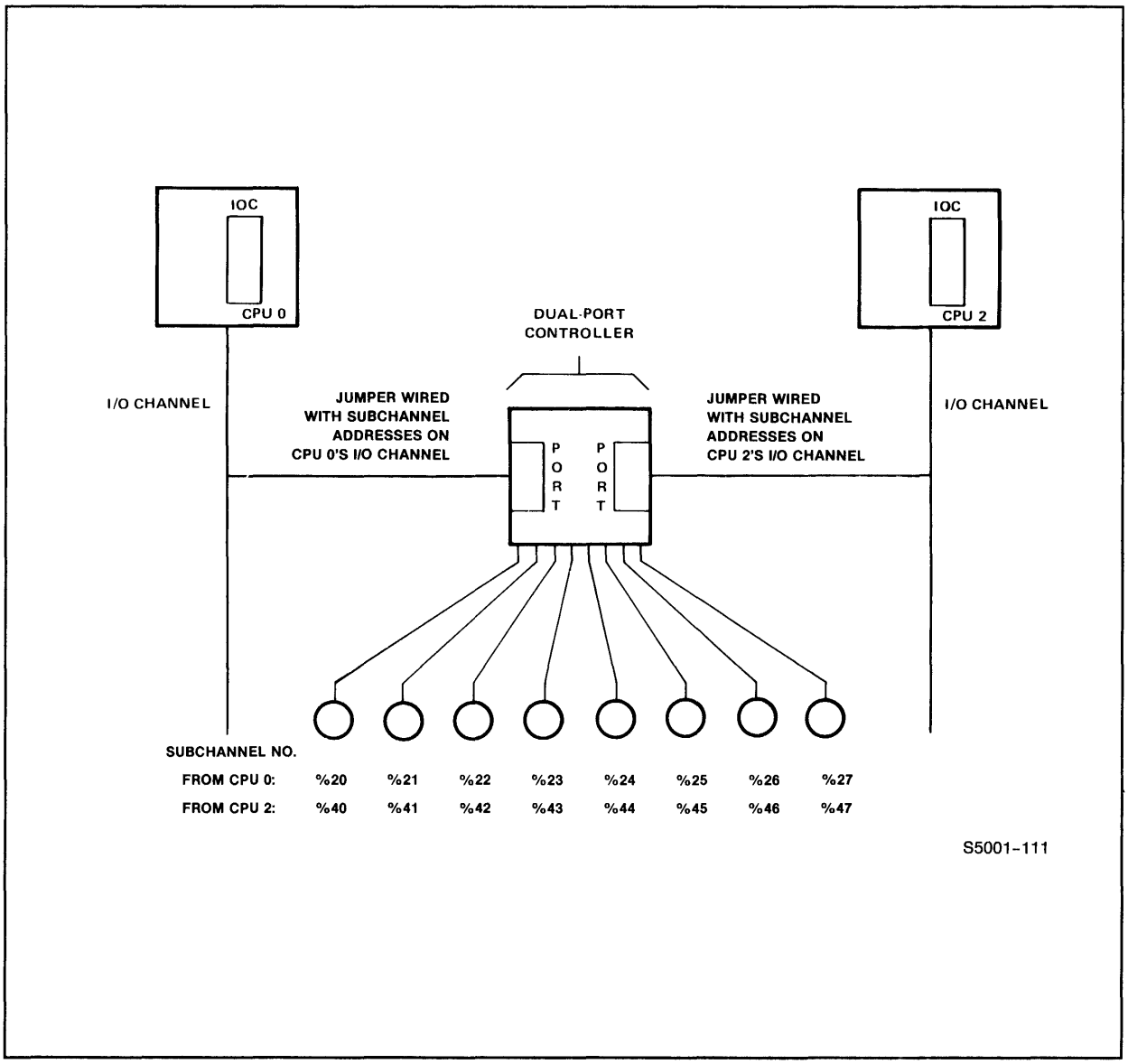


Figure 7-12. Dual-Port Addressing

Although each controller has two ports and is fully capable of communicating through either I/O channel, only one channel is used during normal operation; the other channel, as far as a particular controller is concerned, is not used. The I/O channel through which communication to a particular controller occurs is said to "own" the controller. All I/O transfers (both control and data) occur through the channel owning the controller. This is illustrated in Figure 7-13.

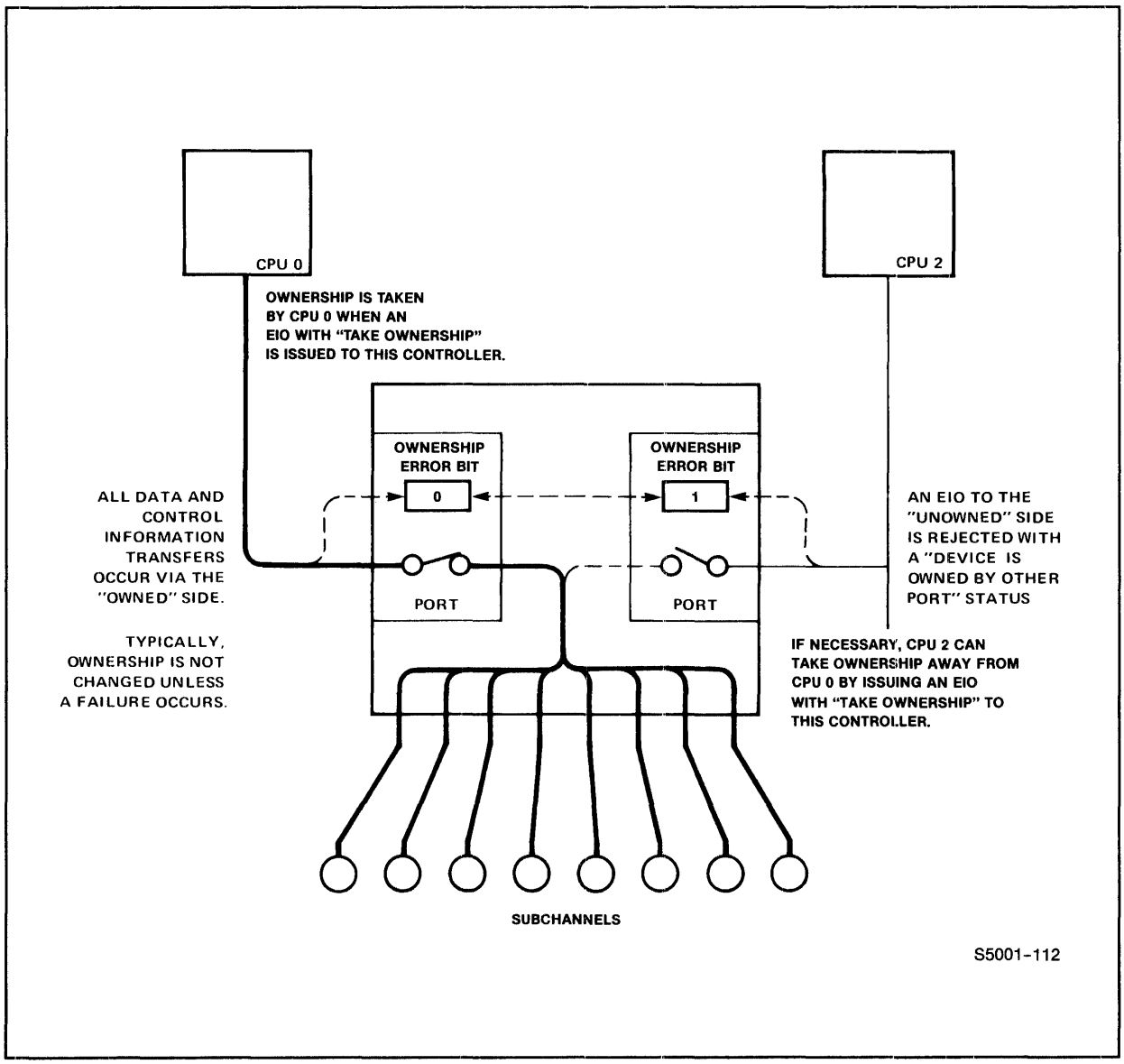


Figure 7-13. I/O Controller Ownership

Each of the two ports in a controller contains a flag bit known as the ownership error bit. The settings of these bits determine the channel from which the controller will accept commands. An operating system configuration parameter specifies which channel is to be the primary channel of communication for a particular controller.

The operating system transfers data only through the owned side. (An attempt to communicate through the unowned side results in the EIO instruction being rejected with an ownership error.) If, during the course of a data transfer, the primary path to the controller (i.e., the primary processor module, channel, or port) becomes inoperable, the operating system generally executes a "take ownership" operation (of an EIO instruction) over the alternate (backup) channel. (One exception: in case of a port failure on a multiple-controller device, the operation is retried using another controller, with no change of ownership.) The ownership bits in the controller switch over to point to the alternate I/O channel. All subsequent data transfers now occur through this channel.

Each port also has two "disable" bits that are separate from its ownership bits. A disable bit, if set to 1, prevents a controller from transmitting information through that port onto an I/O channel. The disable bit is set by an EIO instruction "set disable" command. Normally, this is used by the operating system when a controller performs some unexpected action that could affect the entire channel. The disable bit is associated with a port, so if the malfunction is in one port, normal communication with the controller still occurs via the other port.

I/O Channel Interrupts

A controller signals an interrupt to the IPU when its associated transfer has completed. A controller also interrupts if it is necessary to terminate a transfer prematurely.

When simultaneous interrupts occur on an I/O channel, a priority scheme determines which interrupt is handled first. A subchannel continues to interrupt until it is cleared. Normally, this clearing is done with an IIO or HIIO instruction.

High-Priority I/O

Two levels of interrupt are available on an I/O channel: standard I/O and high-priority I/O. Standard I/O is performed via controllers that interrupt through the SIV entry for standard I/O. Likewise, high-priority I/O is performed via controllers that interrupt through the SIV entry for high-priority I/O. Whether a controller interrupts with standard or high priority is determined by a jumper connection on the controller.

High-priority I/O is used by applications requiring an ultra-fast response time (as in some data communication environments). The operating system never masks off the high-priority interrupt position, thereby ensuring that no matter what is executing in a processor module, a high-priority I/O interrupt will be recognized instantly.

SECTION 8

COLD LOAD

A processor may be initially loaded in one of two ways: from an I/O device (disc) or from another processor using one of the interprocessor buses. The cold-load command may, in turn, be issued from either the control panel switches or the OSP (Operations and Service Processor).

DISC COLD LOAD

To execute a disc cold load from the control panel, the operator sets the Switch Register bits in the following manner: bit 0 to 0; bits 1-6 to the system subvolume of the operating system image to be loaded (SYSnn); bit 7 to 0 unless a LOBUG "boot halt" is requested; and bits 8-15 to the 8-bit subchannel number of the device to be used. After the switches are set, the operator turns the RESET/LOAD key first to the RESET position, then to the LOAD position.

For a disc cold load from the OSP, the operator uses the OSP terminal processor status screen in the following manner: enter the number of the processor to be loaded, then press the F1 function key to select it; then press the F10 function key to reset it. After this, the operator selects the device subchannel number (and SYSnn subvolume, if loading from disc), and presses the F11 function key. The OSP then sends the appropriate cold-load command to the processor. (The equivalent operations can also be peformed using the CPU, SWITCH, RESET, and LOAD commands in the OSP's LOBUG conversational interface.)

The following discussion separately describes a disc cold load for both the NonStop II processor and the NonStop TXP processor.

Disc Cold Load (NonStop II Processor)

In a disc cold-load sequence, the NonStop II processor first executes some microdiagnostics and then performs the following steps:

1. Sets the system data and system code maps (Maps 1 and 3) to map onto physical pages 0 through 63.

2. Sets the ENV, L, and S registers as follows:

```
ENV := %3447;    ! PRIV, DS, and CS bits set;
                 ! V bit set, K bit cleared to identify
                 ! NonStop II CPU
L   := %1000;
S   := %1100;
```

3. Clears the control panel display.

4. Saves the subchannel number from the control panel switches or the OSP in R7 of the Register Stack. The value 1 for an OSP cold load, or the value 0 for a cold load from the switches, is saved in R6.

5. Sets the MASK register to %176000.

6. Sets location %677 in system data to %10777, the value of a BUN -001 (branch to self) machine instruction.

7. Sets the P register to %677.

8. Initializes the IOC entry for the subchannel specified in the switches to the following values:

```
entry.<0:15>  := %100;
entry.<16:31> := %1600;
entry.<32:47> := 0;
entry.<48:63> := 0;
```

9. Takes ownership of the I/O device.

10. Clears pending device interrupts.

11. Issues a cold-load read command to the device to read in the bootstrap program.

12. Begins instruction execution.

The bootstrap program read in in step 11 must perform anything else necessary to load the memory and the Loadable Control Store. This program begins running as soon as location %677 in system

data is overwritten. Its starting conditions are:

```
S    = %1100
P    = %677
ENV  = %3 4 4-- 1  (N and Z bit settings are determined by EIO)
L    = %1000
R7   = value from switches or OSP
R6   = 0 if cold load from switches
       1 if cold load from OSP
R0/R1 = EIO status
Maps 1 and 3 refer to physical pages 0:63
MASK  = %176000
```

Disc Cold Load (NonStop TXP Processor)

The NonStop TXP processor performs the following steps during a disc cold-load sequence:

1. Loads the basic instruction set from DDT prom.

2. Executes microdiagnostics.

3. Sets the invalid bit in the tag word associated with each entry in the data cache and the page table cache.

4. Maps physical pages 0:63 into segment 1. Sets both system code and system data space to segment 1.

5. Sets the ENV, L, and S registers as follows:

```
ENV := %3507;    ! PRIV, DS and CS bits set to 1;
                 ! K bit set, V bit cleared to identify
                 ! NonStop TXP processor
L   := %1000;
S   := %1100;
```

6. Clears the control panel display.

7. Saves the subchannel number from the control panel switches or the OSP in R7 of the Register Stack. Saves a value of 1 in R6 for an OSP cold load, or a value of 0 in R6 for a cold load from the switches.

8. Sets the MASK register to %166000.

9. Null-fills SG[%670:%707] to prevent uncorrectable memory errors due to uninitialized data cache words.

10. Sets SG[%677] equal to %10777, the octal representation of a BUN -001 (branch to self) machine instruction.

11. Sets the P register to %677.

12. Initializes the scratchpad register copy of the IOC entry for the subchannel specified in the switches to the following values:

```
IOC[ device ] := [ %100, %1600, 0, 0 ];
```

13. Takes ownership of the I/O device.

14. Clears pending device interrupts.

15. Issues a cold-load read command to the device to read in the bootstrap program.

16. Checks the Condition Code; halts with SD=%100004 if ENV.Z is not set.

17. Begins instruction execution.

The bootstrap program read in (in step 16) must perform anything else necessary to load the memory and the Loadable Control Store. This program begins running as soon as location %677 is overlaid.

Its starting conditions are:

```
S = %1100
P = %677
ENV = %3 5 0-- 1  (N and Z settings are determined by EIO)
L = %1000
R7 = value from switches or OSP
R6 = 0 if cold load from switches
   = 1 if cold load from OSP
R0/R1 = EIO status
PCACHE[ 1, 0:63 ] refers to physical pages 0:63
SST[ 0:15 ] refer to segments 0, 1, 2, 1, 4, 5, ..., 15
! Both System Code and System Data space set to segment 1 !
MASK = %166000
```

NOTE

A given physical page must be accessed only through a single absolute segment/logical page combination; otherwise, the NonStop TXP processor's code/data cache contents may become inconsistent.

BUS COLD LOAD

For a bus cold load from the control panel, the operator sets Switch Register bit 0 to one and all other bits to zero. Then the operator turns the RESET/LOAD key first to the RESET position, then to the LOAD position.

For a bus cold load from the OSP, the operator loads the processor number to be loaded through a field in the processor status screen, and then selects the processor by pressing function key F1. On the same screen, the operator enters a value of %100000 into the Switch Register field, and resets and loads the processor by pressing F10 and F11. Finally, the operator invokes the operating system's RELOAD program to start the bus cold load. All down CPUs that have been appropriately prepared can be reloaded concurrently.

The following discussion separately describes the bus cold load operation for both NonStop II and NonStop TXP processors.

Bus Cold Load (NonStop II Processor)

In a bus cold-load sequence, the NonStop II processor first executes some microdiagnostics and then performs the following steps:

1. Sets the system data and system code maps (Maps 1 and 3) to map onto physical pages 0 through 63.

2. Sets the ENV, L, and S registers as follows:

```
ENV := %3447;    ! PRIV, DS, and CS bits set;
                 ! V bit set, K bit cleared to identify
                 ! NonStop II CPU
L   := %1000;
S   := %1100;
```

3. Clears the control panel display.

4. Saves the value from the control panel switches or the OSP in R7 of the Register Stack. The value 1 for an OSP cold load, or the value 0 for a cold load from the switches, is saved in R6.

5. Sets the MASK register to %176000.

6. Sets the P register to 0.

7. Reads the bootstrap program and microcode, over one of the buses, into memory starting at SG[0].

8. Begins instruction execution.

The bootstrap program read in in step 7 must perform anything else necessary to load the memory and the Loadable Control Store. This program begins running as soon as 10,530 words have been transferred over the bus to memory. Its starting conditions are:

```
S     = %1100
P     = 0
ENV   = %3447
L     = %1000
R7    = value from switches or OSP
R6    = 0 if cold load from switches
        1 if cold load from OSP
Maps 1 and 3 refer to physical pages 0:63
MASK  = %176000
```

The initial data transfer size allows a transfer of 4096 words of control store, which would occupy 10,240 words of memory, and a bootstrap program of 290 words.

Note that the cold-load bus transfer does not use extended memory addressing. The microcode reads the data from the INQ directly into memory without using the Bus Receive Table (BRT).

Bus Cold Load (NonStop TXP Processor)

The NonStop TXP processor performs the following steps during a bus cold-load sequence:

1. Loads the basic instruction set from DDT prom.

2. Executes microdiagnostics.

3. Sets the invalid bit in the tag word associated with each entry in the data/instruction cache and the page table cache.

4. Maps physical pages 0:63 into segment 1. Sets both system code and system data space to segment 1.

5. Sets the ENV, L, and S registers as follows:

```
ENV := %3507;    ! PRIV, DS and CS bits set to 1;
                 ! K bit set, V bit cleared to identify
                 ! NonStop TXP processor
```

```
L   := %1000;
S   := %1100;
```

6. Clears the control panel display.

7. Saves the value from the control panel switches or the OSP in R7 of the Register Stack. Saves a value of 1 in R6 for an OSP cold load, or a value of 0 in R6 for a cold load from the switches.

8. Sets the MASK register to %166000.

9. Sets the P register to 0.

10. Reads the bootstrap program and microcode, over one of the buses, into memory starting at SG[0].

11. Begins instruction execution.

The "bootstrap" program read in during the cold-load sequence must perform anything else necessary to load the memory and the control store. Its starting conditions are:

```
S      = %1100
P      = %0
ENV    = %3507
L      = %1000
R7     = value from switches or OSP
R6     = 0 if cold load from switches
       = 1 if cold load from OSP
R0/R1  = EIO status
PCACHE[ 1, 0:63 ] refers to physical pages 0:63
SST[ 0:15 ] refer to segments 0, 1, 2, 1, 4, 5, ..., 15
! Both system code and system data space set to segment 1 !
MASK   = %166000
```

Note that the cold-load bus transfer does not use extended memory addressing. The microcode reads the data from the INQ directly into memory without using the Bus Receive Table (BRT).

SECTION 9

INSTRUCTION SET

The instruction sets of the NonStop II and NonStop TXP processors, including the decimal arithmetic and floating-point options, consist of approximately 285 machine instructions. This section provides text descriptions of all these instructions, with the exception of those reserved for operating system use. Diagrams are also included showing the action of some of the more commonly used instructions. To locate the text description for any instruction, refer to the alphabetical listing under "Instructions" in the general index at the back of this manual.

These descriptions assume familiarity with the information presented in Sections 1 through 8. For explanations of terms and concepts mentioned here, refer to the Index to find the appropriate reference.

In addition, Appendixes A and B provide a number of useful reference tables pertaining to the instruction set.

Instructions in this section are categorized by general function and discussed under the following headings:

- 16-Bit Arithmetic
- 32-Bit Signed Arithmetic
- 16-Bit Signed Arithmetic (Register Stack Element)
- Decimal Arithmetic Store and Load (Standard Instructions)
- Decimal Integer Arithmetic (Standard and Optional Instructions)
- Decimal Arithmetic Scaling and Rounding (Standard and Optional Instructions)
- Decimal Arithmetic Conversions (Optional Instructions)
- Floating-Point Arithmetic (Optional Instructions)
- Extended Floating-Point Arithmetic (Optional Instructions)
- Floating-Point Conversions (Optional Instructions)
- Floating-Point Functionals (Optional Instructions)
- Register Stack Manipulation

Boolean Operations
Bit Deposit and Shift
Byte Test
Memory to or from Register Stack
Load and Store Via Address on Register Stack
Branching
Moves, Compares, Scans, and Checksum Computations
Program Register Control
Routine Calls and Returns
Interrupt System
Bus Communication
Input-Output
Miscellaneous
Operating System Functions

NOTE

The instruction descriptions in this section state the conditions under which Overflow is set in the ENV register. If Overflow is set, not part of the results on the stack can be assumed valid. For details on the setting of the Condition Code and Carry bits, refer to Section 4, "Program Environment". Unless otherwise stated, "stack" refers to the Register Stack.

16-BIT ARITHMETIC (Top of Register Stack)

IADD (000210). Integer (signed) Add A to B. A is added to B in integer form. A and B are then deleted from the stack and the sum is pushed onto the stack. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LADD (000200). Logical (unsigned) Add A to B. A and B are added as 16-bit positive integers. A and B are then deleted from the stack and the result pushed on. Carry is set if the addition overflows bit 0. Condition Code is set.

ISUB (000211). Integer (signed) Subtract A from B. A is subtracted from B in integer form. A and B are deleted and the difference is pushed onto the stack. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LSUB (000201). Logical (unsigned) Subtract A from B. A is subtracted from B logically. A and B are then deleted from the stack and the result pushed on. Carry is set if A is less than or equal to B. Condition Code is set.

IMPY (000212). Integer (signed) Multiply A times B. B is multiplied by A in integer form. A and B are deleted from the stack and the result pushed on. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LMPY (000202). Logical (unsigned) Multiply A times B. A and B are multiplied as 16-bit positive integers. A and B are then replaced by the doubleword result, with the least significant half in A. Overflow is implicitly cleared. Condition Code is set.

IDIV (000213). Integer (signed) Divide B by A. B is divided by A in integer form. A and B are deleted from the stack and the result pushed on. Overflow is set if the divisor is zero, or if the result is greater than 32767 or less than -32768. Condition Code is set.

LDIV (000203). Logical (unsigned) Divide CB by A, leaving the remainder in B. The 32-bit positive integer in C and B is divided by the 16-bit positive integer in A. The divisor and dividend are deleted from the stack, the remainder is pushed onto the stack (B), and the quotient is pushed onto the stack (A). Overflow is set if the original C is greater than or equal to the original A. Condition Code is set.

INEG (000214). Integer (signed) Negate A. A is converted to its two's complement form. Overflow is set if the original operand was -32768. Condition Code is set.

LNEG (000204). Logical (unsigned) Negate A. A is converted to its two's complement. Carry is set if the original value of A is zero. Condition Code is set.

ICMP (000215). Integer (signed) Compare B with A. B is compared to A in integer form and the Condition Code set accordingly. A and B are then deleted from the stack.

LCMP (000205). Logical (unsigned) Compare B with A. B is logically compared to A and the Condition Code set accordingly. A and B are then deleted from the stack.

CMPI (001---). Compare A with Immediate Operand. The Condition Code is set as a result of the 16-bit integer comparison of A and the immediate operand. A is then deleted from the stack. Examples of the use of immediate operands are shown in Figure 9-1.

ADDI (104---). Add Immediate Operand to A. The immediate operand is added to A in integer form. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LADI (003---). Logical (unsigned) Add Immediate Operand to A. The immediate operand is pushed onto the stack, with the sign bit propagating into the high order bits. Then A and B are added in 16-bit unsigned integer form. A and B are then both deleted from the stack and the result pushed on. Carry is set if the addition overflows bit 0. Condition Code is set.

32-BIT SIGNED ARITHMETIC

DADD (000220). Double Add DC to BA. The two doubleword integers contained in DC and BA are added in doubleword integer form. Both operands are then deleted, and the doubleword result is pushed onto the stack. Overflow is set if the result is greater than (2**31)-1 or less than -(2**31). Carry can be set, and Condition Code is set on the result.

DSUB (000221). Double Subtract BA from DC. The doubleword integer contained in BA is subtracted in doubleword integer form from the doubleword integer in DC. Both operands are then deleted, and the result is pushed onto the stack. Overflow is

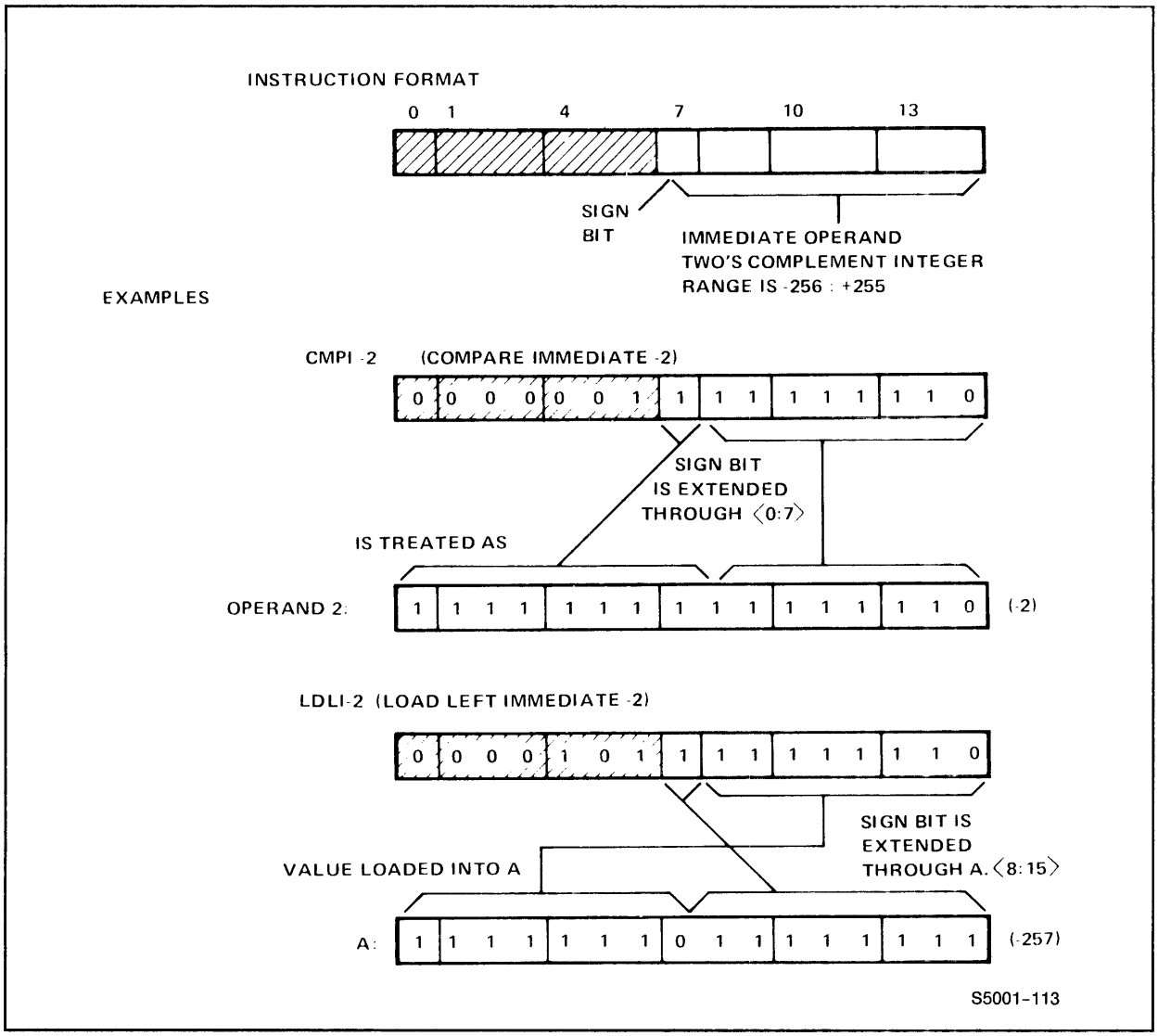


Figure 9-1. Immediate Operand

set if the result is greater than (2**31)-1 or less than -(2**31). Carry can be set, and Condition Code is set on the result.

DMPY (000222). Double Multiply DC by BA. The doubleword integer contained in DC is multiplied in doubleword integer form by the doubleword integer in BA. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than (2**31)-1 or less than -(2**31). Carry can be set, and Condition Code is set on the result.

DDIV (000223). Double Divide DC by BA. The doubleword integer contained in DC is divided in doubleword integer form by the doubleword integer in BA. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than (2**31)-1 or less than -(2**31), or if the divisor (BA) is zero. Carry can be set, and Condition Code is set on the result.

DNEG (000224). Double Negate BA. The doubleword integer contained in BA is replaced with its two's complement. Overflow is set if the original operand was -(2**31). Carry can be set, and Condition Code is set on the result.

DCMP (000225). Double Compare DC with BA. The Condition Code in the ENV register is set as a result of the doubleword integer comparison of DC and BA. Both operands are then deleted from the stack.

DTST (000031). Double Test BA. The Condition Code is set according to the contents of the doubleword contained in BA.

CDI (000307). Convert Double to Integer. The doubleword integer in BA is converted to a singleword integer by copying the contents of A into B and deleting A. Overflow is set if the doubleword quantity is greater than 32767 or less than -32768.

CID (000327). Convert Integer to Double. The singleword integer in A is extended to a doubleword quantity on the top of the Register Stack. A is copied into H, and then A is filled with zeros if A was positive, or ones if A was negative; the Register Pointer is incremented to give the result in BA.

MOND (000001). Minus One Double. A doubleword minus one is pushed onto the top of the Register Stack (BA). Condition Code is set.

ZERD (000002). Zero Double. A doubleword zero is pushed onto the top of the Register Stack (BA). Condition Code is set.

ONED (000003). One Double. A doubleword of one is pushed onto the top of the Register Stack (BA). Condition Code is set.

16-BIT SIGNED ARITHMETIC (REGISTER STACK ELEMENT)

NOTE

For binary coding details of the first four instructions that follow (ADRA, SBRA, ADAR, SBAR), refer to Table A-7 in Appendix A. For ADXI, refer to Table A-4.

ADRA (00014-). Add Register to A. The contents of the register pointed to by the Register field of the instruction are added in integer form to register A. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

SBRA (00015-). Subtract Register from A. The contents of the register pointed to by the Register field of the instruction are subtracted in integer form from register A. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

ADAR (00016-). Add A to a Register. A is added in signed integer form to the register pointed to by the Register field of the instruction. A is deleted from the stack. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

SBAR (00017-). Subtract A from a Register. A is subtracted in signed integer form from the register pointed to by the Register field of the instruction. A is deleted from the stack. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

ADXI (104---). Add Immediate Operand to an Index Register. The immediate operand is added in signed integer form to the contents of the index register specified by "x" field of the instruction. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set; Condition Code is set on the result.

DECIMAL ARITHMETIC STORE AND LOAD (STANDARD INSTRUCTIONS)

NOTE

For binary coding details of the following two instructions, refer to Table A-8 in Appendix A.

QST (00023-). Quadruple Store. The quadrupleword operand contained in EDCB is stored in the effective memory location indicated by A plus 4 times the index value. No indexing occurs for coding 000230. For code 000231, 000232, or 000233, indexing for the effective address uses register R[5], R[6], or R[7], respectively. The quadrupleword operand and A are then deleted from the stack.

QLD (00023-). Quadruple Load. The quadrupleword operand contained in the effective memory location indicated by A plus 4 times the index value is fetched. A is deleted, and the fetched quadrupleword is pushed onto the stack. No indexing occurs for coding 000234. For code 000235, 000236, or 000237, indexing for the effective address uses register R[5], R[6], or R[7], respectively. Condition Code is set on the loaded quadrupleword.

DECIMAL INTEGER ARITHMETIC (STANDARD AND OPTIONAL INSTRUCTIONS)

QADD (000240). Quadruple Add. The two quadrupleword integers contained in HGFE and DCBA are added in quadrupleword integer form. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Carry can be set, and Condition Code is set on the result. (This is a standard instruction.)

QSUB (000241). Quadruple Subtract. The quadrupleword integer contained in DCBA is subtracted in quadruple-length integer form from the quadrupleword integer in HGFE. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Carry can be set, and Condition Code is set on the result. (This is a standard instruction.)

QMPY (000242). Quadruple Multiply. The quadrupleword integer contained in HGFE is multiplied in quadrupleword integer form by the quadrupleword integer in DCBA. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Carry can be set, and Condition Code is set on the result. (This is an optional instruction.)

QDIV (000243). Quadruple Divide. The quadrupleword integer contained in HGFE is divided in quadrupleword integer form by the quadrupleword integer in DCBA. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the divisor (DCBA) is zero. Condition Code is set. (This is an optional instruction.)

QNEG (000244). Quadruple Negate. The quadrupleword integer contained is DCBA is replaced with its two's complement. Overflow is set if the original operand was -(2**63). Condition Code is set on the result. (This is an optional instruction.)

QCMP (000245). Quadruple Compare. The Condition Code in the Environment Register is set according to the quadruple integer comparison of HGFE (operand 1) and DCBA (operand 2). (See Table A-3 for Condition Code settings; the "a" states apply for compares.) Both operands are then deleted from the stack. (This is an optional instruction.)

DECIMAL ARITHMETIC SCALING AND ROUNDING (STANDARD AND OPTIONAL INSTRUCTIONS)

NOTE

For binary coding details of the following three instructions, refer to Table A-8 in Appendix A.

QUP (00025-). Quadruple Scale Up. The operand value in DCBA is multiplied by a specified power of ten (1, 2, 3, or 4), and the new value replaces the former contents of DCBA. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result. (This is a standard instruction.)

QDWN (00025-). Quadruple Scale Down. The operand value in DCBA is divided by a specified power of ten (1, 2, 3, or 4), and the new value replaces the former contents of DCBA. Condition Code is set, and the Overflow bit is cleared. (This is a standard instruction.)

QRND (000263). Quadruple Round. Five is added to the operand in DCBA if the operand is positive (-5 is added if negative), and the result is divided by 10. The new value replaces the former contents of DCBA. Condition Code is set, and the Overflow bit is cleared. (This is an optional instruction.)

DECIMAL ARITHMETIC CONVERSIONS (OPTIONAL INSTRUCTIONS)

CQI (000264). Convert Quad to Integer. The four-word value in DCBA is converted to an integer by extracting the least significant word. DCBA is deleted, and the integer result is pushed onto the stack. Overflow is set if the operand was greater than 32767 or less than -32768.

CQL (000246). Convert Quad to Logical. The four-word value in DCBA is converted to a logical value by extracting the least significant word. DCBA is deleted, and the integer result is pushed onto the stack. Overflow is set if the operand was greater than 65535.

CQD (000247). Convert Quad to Double. The four-word value in DCBA is converted to a doubleword by extracting the least significant two words. DCBA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the operand was greater than (2**31)-1 or less than -(2**31).

CQA (000260). Convert Quad to ASCII. The absolute value of the binary-coded quadrupleword integer in FEDC is converted to a string of ASCII-coded digits (decimal base), and the resulting string is stored in the memory space defined by a starting byte address in B and a byte count in A. If the conversion results in a truncation of leading digits, overflow is set. Condition Code is set on the original value.

CIQ (000266). Convert Integer to Quad. The singleword integer in A is extended to a quadrupleword quantity, filling the most significant three words with zeros if A was positive, or ones if A was negative. A is deleted, and the quadrupleword result is pushed onto the stack.

CLQ (000267). Convert Logical to Quad. The singleword logical quantity in A is extended to a quadrupleword quantity, filling the most significant three words with zeros. A is deleted, and the quadrupleword result is pushed onto the stack.

CDQ (000265). Convert Double to Quad. The doubleword integer in BA is extended to a quadrupleword quantity, filling the most significant two words with zeros if B is positive, or ones if B is negative. BA is deleted, and the quadrupleword result is pushed onto the stack.

CAQ (000262). Convert ASCII to Quad. A string of 7-bit ASCII-coded digits in memory, defined by a starting byte address in B and a byte count in A, is converted to a binary-coded quadrupleword integer. The quadrupleword result is pushed onto the stack. If a nondigit ASCII code is encountered, only the preceding digits are converted, and CCG indicates that only part of the string was converted; CCE indicates that the entire string was converted. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). If overflow is set, the value in DCBA is undefined.

CAQV (000261). Convert ASCII to Quad with Initial Value. A string of ASCII-coded digits in memory, defined by a starting byte address in F and a byte count in E, is converted to a binary-coded quadrupleword integer in DCBA. DCBA contains an initial value (greater than or equal to zero) which is multiplied by 10, providing a high-order value to which the converted value is added to produce the result in DCBA. If a nondigit ASCII code is encountered, only the preceding digits are converted, and CCG indicates that only part of the string was converted; CCE indicates that the entire string was converted. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). If overflow is set, the value in DCBA is undefined.

FLOATING-POINT ARITHMETIC (OPTIONAL INSTRUCTIONS)

NOTE

For the range of floating-point numbers, refer to "Number Representations" in Section 3.

FADD (000270). Floating-Point Add. The floating-point quantities in DC and BA are added in floating-point form. Both operands are deleted, and the two-word result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

FSUB (000271). Floating-Point Subtract. The floating-point quantity in BA is negated, and then DC and BA are added in floating-point form. Both operands are deleted, and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating- point numbers. Condition Code is set on the result.

FMPY (000272). Floating-Point Multiply. The floating-point quantities in DC and BA are multiplied in floating-point form. Both operands are deleted, and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

FDIV (000273). Floating-Point Divide. The floating-point quantity in DC is divided in floating-point form by the floating-point quantity in BA. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

FNEG (000274). Floating-Point Negate. The floating-point quantity in BA (if not zero) is negated. The sign of BA is reversed from positive to negative or negative to positive, and the Condition Code reflects the final state of the sign (see Table A-3).

FCMP (000275). Floating-Point Compare. The Condition Code is set according to the comparison of DC (operand 1) with BA (operand 2). (See Table A-3 for Condition Code settings; the "a" states apply for comparisons.) Both operands are then deleted from the stack.

EXTENDED FLOATING-POINT ARITHMETIC (OPTIONAL INSTRUCTIONS)

NOTE

For the range of extended floating-point numbers, refer to "Number Representations" in Section 3.

EADD (000300). Extended Add. The extended floating-point quantities in HGFE and DCBA are added in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

ESUB (000301). Extended Subtract. The extended floating-point quantity in HGFE is negated, and then HGFE and DCBA are added in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

EMPY (000302). Extended Multiply. The extended floating-point quantities in HGFE and DCBA are multiplied in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

EDIV (000303). Extended Divide. The extended floating-point quantity in HGFE is divided in extended floating-point form by the extended floating-point quantity in DCBA. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

ENEG (000304). Extended Negate. The extended floating-point quantity in DCBA (if not zero) is negated. The sign of DCBA is reversed from positive to negative or negative to positive. Overflow is cleared, and the Condition Code reflects the final state of the sign.

ECMP (000305). Extended Compare. The Condition Code is set according to the comparison of HGFE (operand 1) with DCBA (operand 2). Both operands are then deleted from the stack.

FLOATING-POINT CONVERSIONS (OPTIONAL INSTRUCTIONS)

CEF (000276). Convert Extended to Floating. The four-word floating-point quantity in DCBA is converted to a two-word floating-point quantity. DCBA is deleted, and the two-word result is pushed onto the stack.

CEFR (000277). Convert Extended to Floating, Rounded. The four-word floating-point quantity in DCBA is converted to a two-word floating-point quantity. The new quantity is rounded according to the contents of truncated bit 7 of C. DCBA is deleted, and the two-word result is pushed onto the stack.

CFI (000311). Convert Floating to Integer. The floating-point quantity in BA is converted to a singleword signed integer. A is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFIR (000310). Convert Floating to Integer, Rounded. The floating-point quantity in BA is converted to a singleword signed integer, with rounding according to the contents of the most significant fractional bit. A is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFD (000312). Convert Floating to Double. The floating-point quantity in BA is converted to a doubleword signed integer in BA.

Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CFDR (000313). Convert Floating to Double, Rounded. The floating-point quantity in BA is converted to a doubleword signed integer in BA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CED (000314). Convert Extended to Double. The extended floating-point quantity in DCBA is converted to a doubleword signed integer. BA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CEDR (000315). Convert Extended to Double, Rounded. The extended floating-point quantity in DCBA is converted to a doubleword signed integer, with rounding according to the contents of the most significant fractional bit. BA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CEI (000337). Convert Extended to Integer. The extended floating-point quantity in DCBA is converted to a singleword signed integer. CBA is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CEIR (000316). Convert Extended to Integer, Rounded. The extended floating-point quantity in DCBA is converted to a singleword signed quantity, with rounding according to the contents of the most significant fractional bit. CBA is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFQ (000320). Convert Floating to Quadruple. The floating-point quantity in BA is converted to a quadrupleword integer in DCBA. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CFQR (000321). Convert Floating to Quadruple, Rounded. The floating-point quantity in BA is converted to a quadrupleword integer in DCBA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CEQ (000322). Convert Extended to Quadruple. The extended floating-point quantity in DCBA is converted to a quadrupleword integer in DCBA. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CEQR (000323). Convert Extended to Quadruple, Rounded. The extended floating-point quantity in DCBA is converted to a quadrupleword integer in DCBA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CFE (000325). Convert Floating to Extended. The floating-point quantity in BA is converted to an extended floating-point quantity. BA is deleted, and the four-word result is pushed onto the stack.

CIF (000331). Convert Integer to Floating. The signed integer in A is converted to a floating-point quantity. A is deleted, and the two-word result is pushed onto the stack.

CDF (000306). Convert Double to Floating. The doubleword signed integer in BA is converted to a floating-point quantity in BA, with truncation if the result exceeds 23 significant bits.

CDFR (000326). Convert Double to Floating, Rounded. The doubleword signed integer in BA is converted to a floating-point quantity in BA, with rounding if the result exceeds 23 significant bits.

CQF (000324). Convert Quadruple to Floating. The quadrupleword signed integer in DCBA is converted to a floating-point quantity, with truncation if the result exceeds 23 significant bits. DCBA is deleted, and the two-word result is pushed onto the stack.

CQFR (000330). Convert Quadruple to Floating, Rounded. The quadrupleword signed integer in DCBA is converted to a floating-point quantity, with rounding if the result exceeds 23 significant bits. DCBA is deleted, and the two-word result is pushed onto the stack.

CIE (000332). Convert Integer to Extended. The signed integer in A is converted to an extended floating-point quantity. A is deleted, and the four-word result is pushed onto the stack.

CDE (000334). Convert Double to Extended. The doubleword signed integer in BA is converted to an extended floating-point quantity. BA is deleted, and the four-word result is pushed onto the stack.

CQE (000336). Convert Quadruple to Extended. The quadrupleword signed integer in DCBA is converted to an extended floating-point quantity in DCBA, with truncation if the result exceeds 55 significant bits.

CQER (000335). Convert Quadruple to Extended, Rounded. The quadrupleword signed integer in DCBA is converted to an extended floating-point quantity in DCBA, with rounding if the result exceeds 55 significant bits.

FLOATING-POINT FUNCTIONALS (OPTIONAL INSTRUCTIONS)

IDX1 (000344). Calculate Index, 1 Dimension. For a one-dimensional array, IDX1 compares the subscript value in B against lower and upper bounds in a two-word table in the current code segment starting at the address specified in A. If the value is in bounds, the element offset value is computed and is stored in register R[7]. If the subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. BA is then deleted.

IDX2 (000345). Calculate Index, 2 Dimensions. For a two-dimensional array, IDX2 compares the subscript values in B and C against lower and upper bounds in a four-word table in the current code segment starting at the address in A. If the values are in bounds, the element offset value is computed and stored in register R[7]. If a subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. CBA is then deleted.

IDX3 (000346). Calculate Index, 3 Dimensions. For a three-dimensional array, IDX3 compares the subscript values in B, C, and D against lower and upper bounds in a six-word table in the current code segment starting at the address in A. If the values are in bounds, the element offset value is computed and stored in register R[7]. If any subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. DCBA is then deleted.

IDXP (000347). Calculate Index, Code Space. For an n-dimensional array, IDXP compares the subscript values in n stack registers (B, C, D, etc.) against lower and upper bounds in a table in the current code segment (2n words) specified by a starting address in A. (The first word of the table in memory is the number of dimensions.) If the values are in bounds, the element offset value is computed and stored in register R[7]. If any subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. All stack data used is deleted.

IDXD (000317). Calculate Index, Data Space. For an n-dimensional array, IDXD compares the subscript values in n stack registers (B, C, D, etc.) against lower and upper bounds in

a table in the current data segment (2n words) specified by a starting address in A. (The first word of the table in memory is the number of dimensions.) If the values are in bounds, the element offset value is computed and stored in register R[7]. If any subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. All stack data used is deleted.

REGISTER STACK MANIPULATION

EXCH (000004). Exchange A and B. A and B of the Register Stack are interchanged. Condition Code is set on the result in A.

DXCH (000005). Double Exchange BA with DC. The doubleword contained in DC is interchanged with the doubleword contained in BA. Condition Code is set on the result in BA.

DDUP (000006). Double Duplicate BA in DC. The doubleword in the top two registers of the stack is duplicated by pushing a copy of it onto the Register Stack. Condition Code is set.

NOTE

For binary coding details of the following three instructions (STAR, NSAR, LDRA), refer to Table A-7 in Appendix A.

STAR (00011-). Store A in a Register. The A Register contents are stored in the register pointed to by the Register field of the instruction. A is then deleted from the stack.

NSAR (00012-). Non-destructive Store A into a Register. The A Register is stored in the register pointed to by the Register field of the instruction.

LDRA (00013-). Load A from a Register. The contents of the register pointed to by the Register field of the instruction are pushed onto the stack. Condition Code is set.

NOTE

For binary coding details of the following three instructions (LDI, LDXI, LDLI), refer to Table A-4 in Appendix A.

LDI (100---). Load Immediate Operand into A. The immediate operand is pushed onto the stack, with the sign bit propagating into the high-order bits. Condition Code is set.

LDXI (10----). Load Index Register with Immediate Operand. The index register specified by the "x" field of the instruction is loaded with the immediate operand, and the sign bit propagates into the high- order bits. Condition Code is set.

LDLI (005---). Load Left Immediate Operand into bits 0:7 of A. The immediate operand, shifted left eight places, is loaded into A, with the sign bits propagating into the low-order bits of A. Condition Code is set.

BOOLEAN OPERATIONS

Figure 9-2 illustrates the fundamental principles of boolean operations as performed by four of the instructions. Figure 9-3 shows equivalent operations as performed on immediate operands.

LAND (000010). Logical AND A with B. A and B are logically ANDed. The two words are deleted from the stack and the result pushed on. Condition Code is set.

LOR (000011). Logical OR A with B. A and B are merged by a logical inclusive OR. A and B are deleted and the result pushed onto the stack. Condition Code is set.

XOR (000012). Logical Exclusive OR A with B. The two words in A and B of the Register Stack are combined by a logical exclusive OR. The two words are then deleted and the result is pushed onto the stack. Condition Code is set.

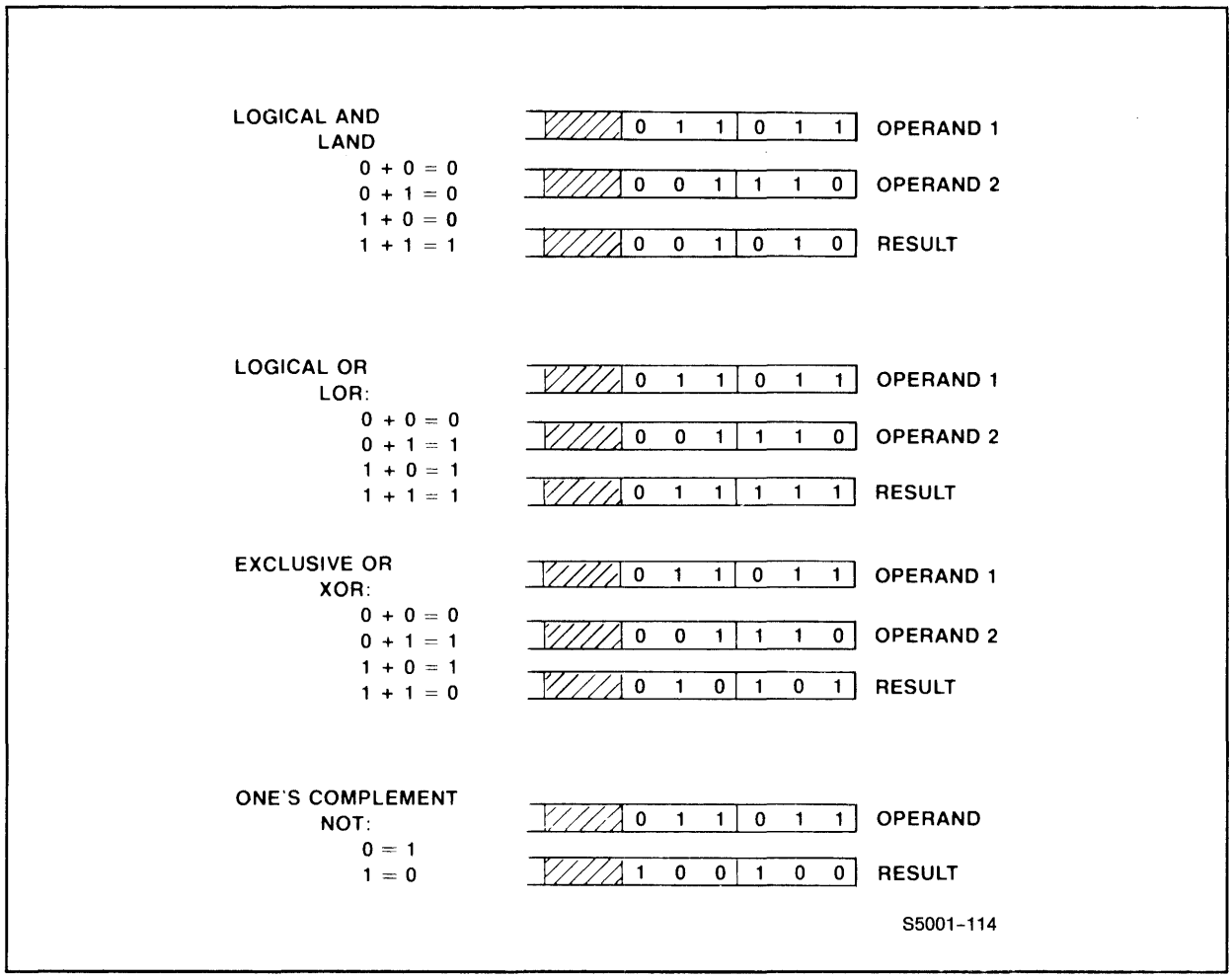


Figure 9-2. Boolean Operations

NOT (000013). One's Complement A. The word contained in Register A of the stack is converted to its one's complement. Condition Code is set.

NOTE

For binary coding details of the following four instructions (ORRI, ORLI, ANRI, ANLI), refer to Table A-4 in Appendix A.

ORRI (004---). OR Right Immediate Operand with A. The 8-bit immediate operand is merged with the A Register by a logical inclusive OR. The sign bit is not propagated, but is actually part of the instruction; see Figure 9-3. Condition Code is set.

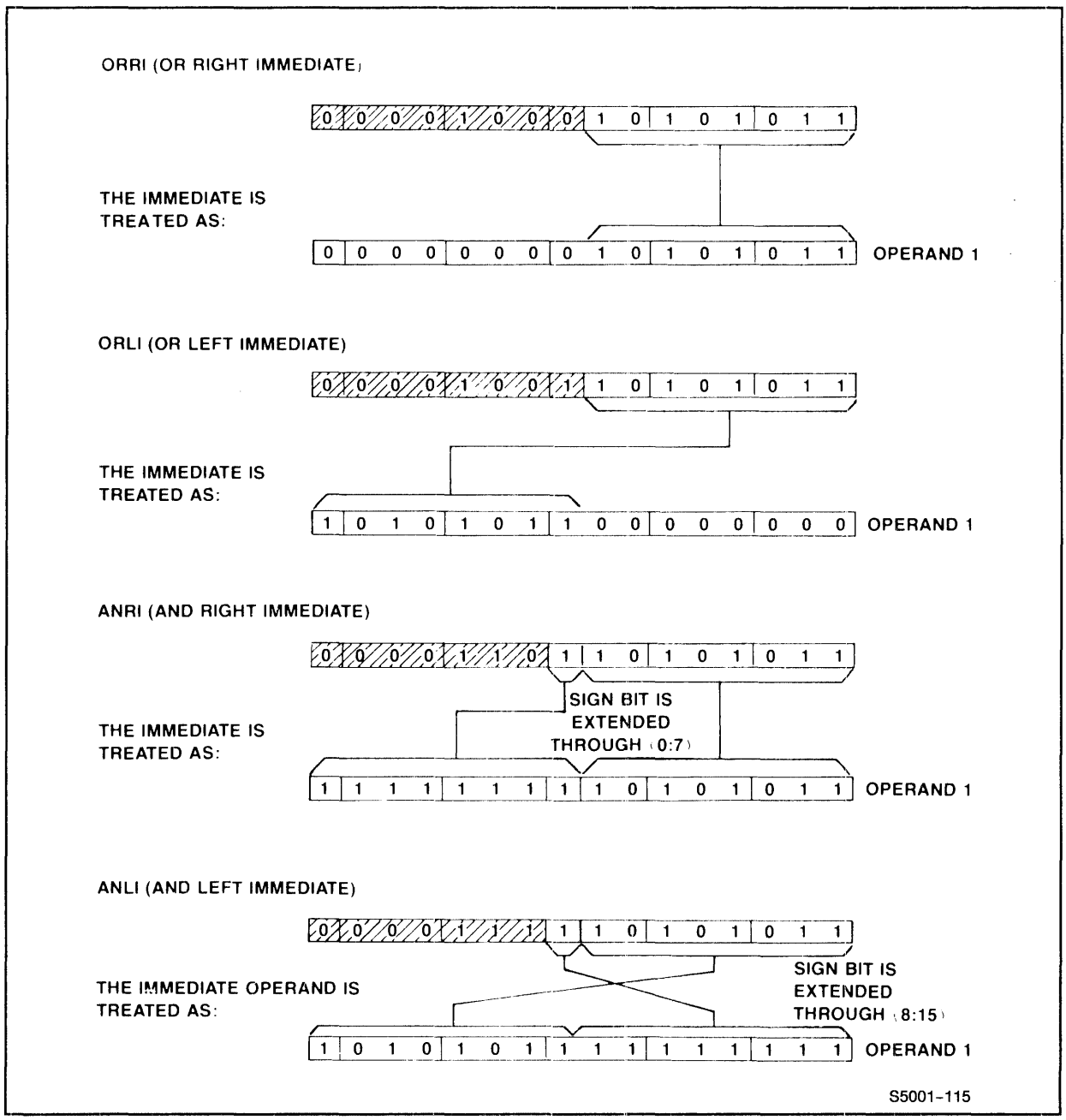


Figure 9-3. Boolean Instructions with Immediate Operands

ORLI (004---). OR Left Immediate Operand with A. The 8-bit immediate operand is shifted left eight places and merged with A by a logical inclusive OR. The sign bit is not propagated, but is actually part of the instruction; see Figure 9-3. Condition Code is set.

ANRI (006---). AND Right Immediate Operand to A. The 8-bit immediate operand is extended to 16 bits by propagating the sign into the high-order bits, and the resulting integer is logically ANDed to A; see Figure 9-3. Condition Code is set.

ANLI (007---). AND Left Immediate Operand with A. The 8-bit immediate operand is shifted left eight places, the sign bit is propagated into the low-order bits, and the resulting integer is logically ANDed to A; see Figure 9-3. Condition Code is set.

BIT DEPOSIT AND SHIFT

DPF (000014). Deposit Field in A. This instruction combines the words contained in registers A and C of the stack as a function of a mask word contained in register B of the stack. A logical OR operation is performed on the logical AND of B and C and the logical AND of not B and A, so that all bits in C corresponding to ones in B are deposited into corresponding bits in A. The original three words are deleted from the stack and the result pushed onto the stack. Condition Code is set. An example of this operation is shown in Figure 9-4.

LLS (0300--). Logical (unsigned) Left Shift. If the Shift Count field is zero, the word contained in B is shifted left by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted left by that number. Condition Code is set. Figure 9-5 presents a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

DLLS (1300--). Double Logical (unsigned) Left Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted left by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted left by that number. Condition Code is set.

LRS (0301--). Logical (unsigned) Right Shift. If the Shift Count field is zero, the word contained in B is shifted right by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted right by that number. Condition Code is set.

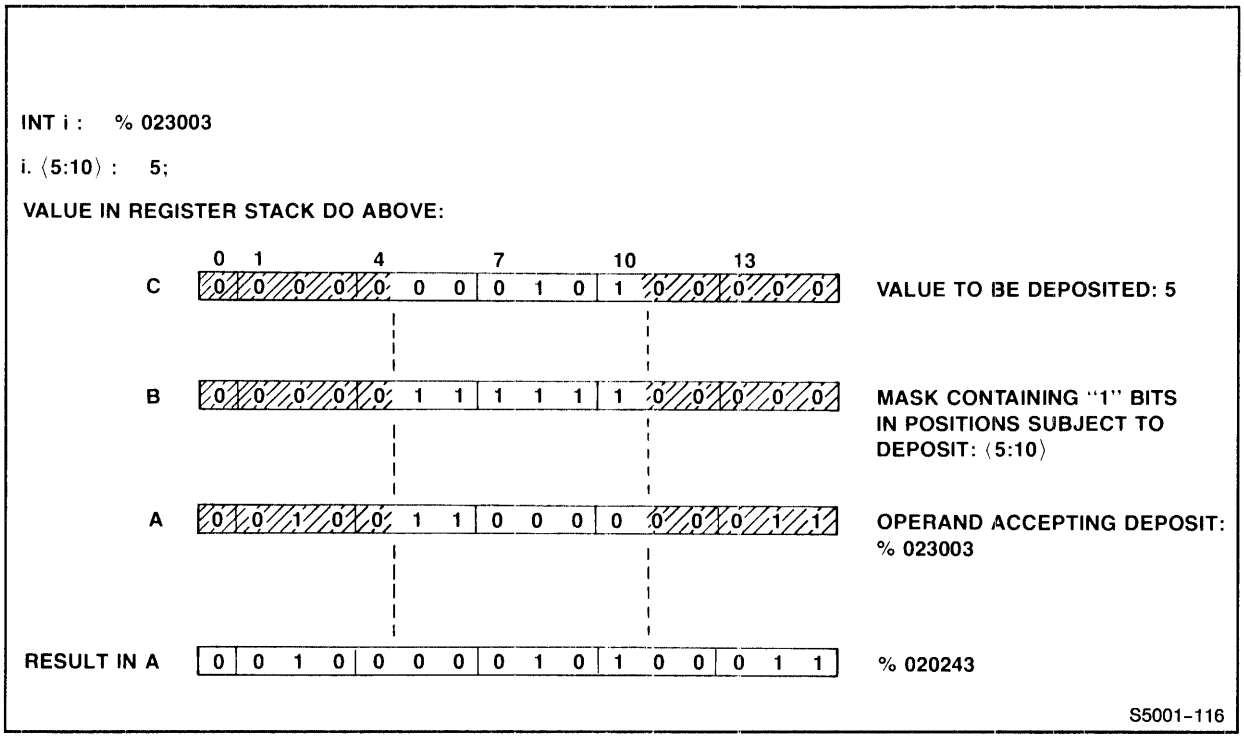


Figure 9-4. DPF Instruction Example

DLRS (1301--). Double Logical (unsigned) Right Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted right by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted right by that number. Condition Code is set.

ALS (0302--). Arithmetic (signed) Left Shift. If the Shift Count field is zero, the word contained in B is shifted left preserving the sign bit by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted left, preserving the sign bit, by that number. Condition Code is set.

DALS (1302--). Double Arithmetic (signed) Left Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted left, preserving the sign bit, by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted left, preserving the sign bit, by that number. Condition Code is set.

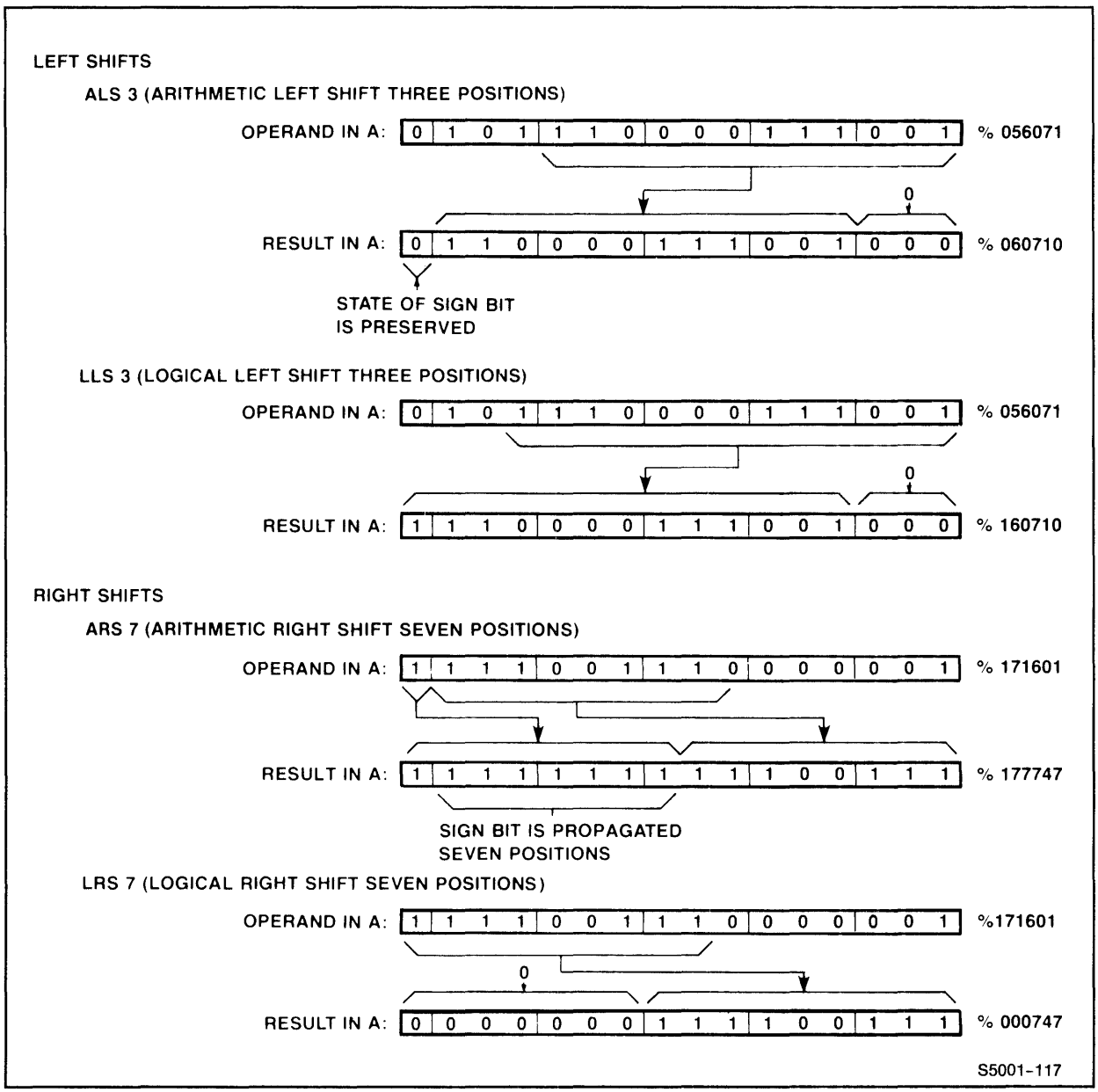


Figure 9-5. Arithmetic Versus Logical Shifts

ARS (0303--). Arithmetic (signed) Right Shift. If the Shift Count field is zero, the word contained in B is shifted right, propagating the sign bit, by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted right, propagating the sign bit, by that number. Condition Code is set.

DARS (1303--). Double Arithmetic (signed) Right Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted right, propagating the sign bit, by the count (modulo %400) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted right, propagating the sign bit, by that number. Condition Code is set.

BYTE TEST

BTST (000007). Byte Test A. The Condition Code is set on the value of the test byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character. A is deleted after the test.

MEMORY TO OR FROM REGISTER STACK

NOTE

For binary coding details of the first twelve instructions below (LWP through ADM), refer to Table A-3 in Appendix A.

LWP (-2----). Load Word from Program (Current Code Segment) into A. The contents of the address which is computed as a function of displacement (a signed 8-bit value), and optionally indexing and indirection, are pushed onto the Register Stack. Condition Code is set on the loaded word. Figure 9-6 illustrates the addressing operations for the LWP instruction.

LBP (-2-4--). Load Byte from Program (Current Code Segment) into A. The contents of the P-relative byte address which is computed as a function of displacement (a signed 8-bit value), and optionally indexing and indirection, are pushed onto the Register Stack. The high-order byte is set to zero. If the P Register currently indicates an address in the upper half of the code segment (bit 0 of P = 1), %100000 is added to the computed address, so that the address will always be relative to whichever half of the segment P currently indicates. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character. Figure 9-7 illustrates the addressing operations for the LBP instruction, assuming addresses in the first half of the code segment.

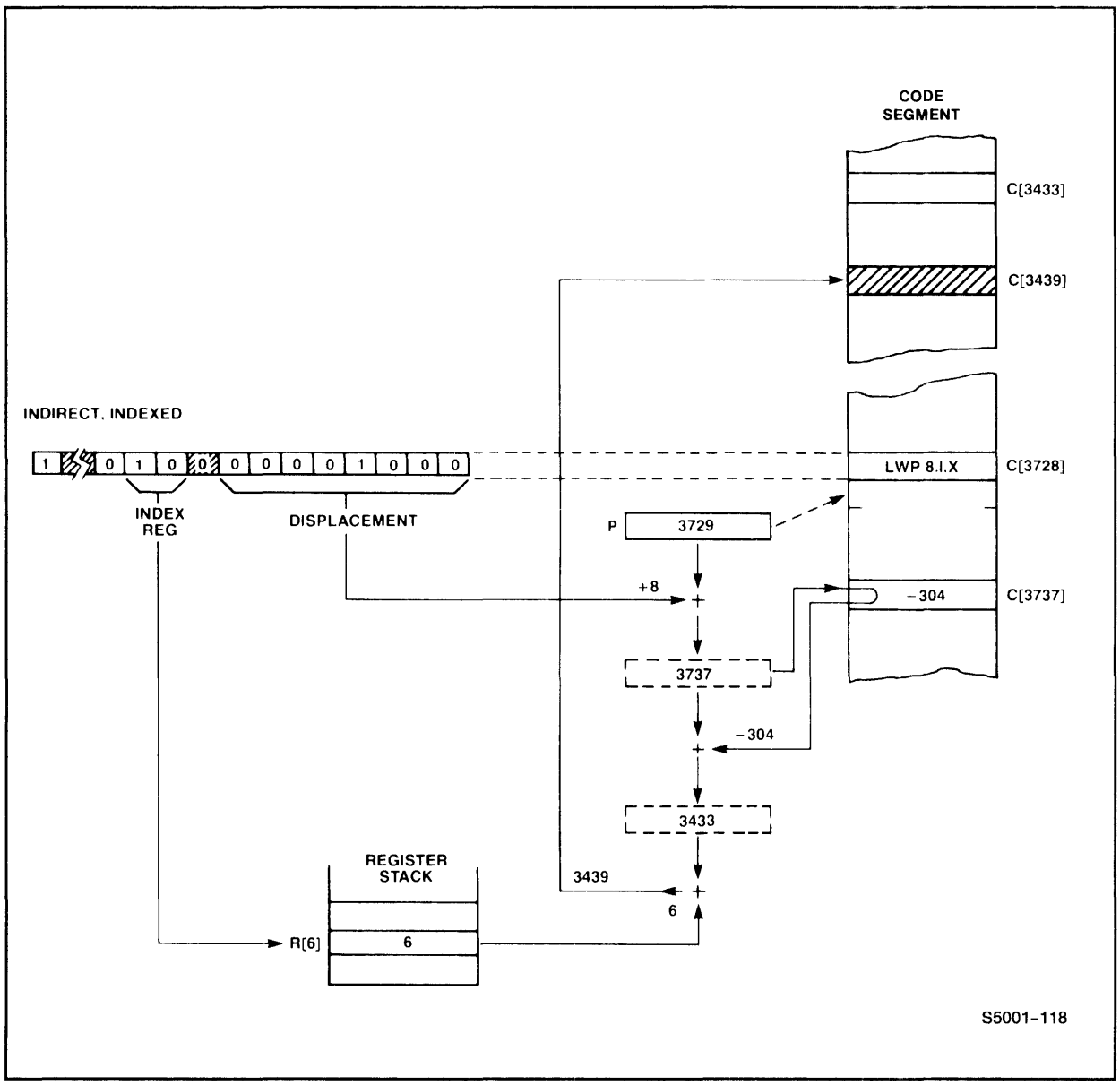


Figure 9-6. LWP Instruction Addressing

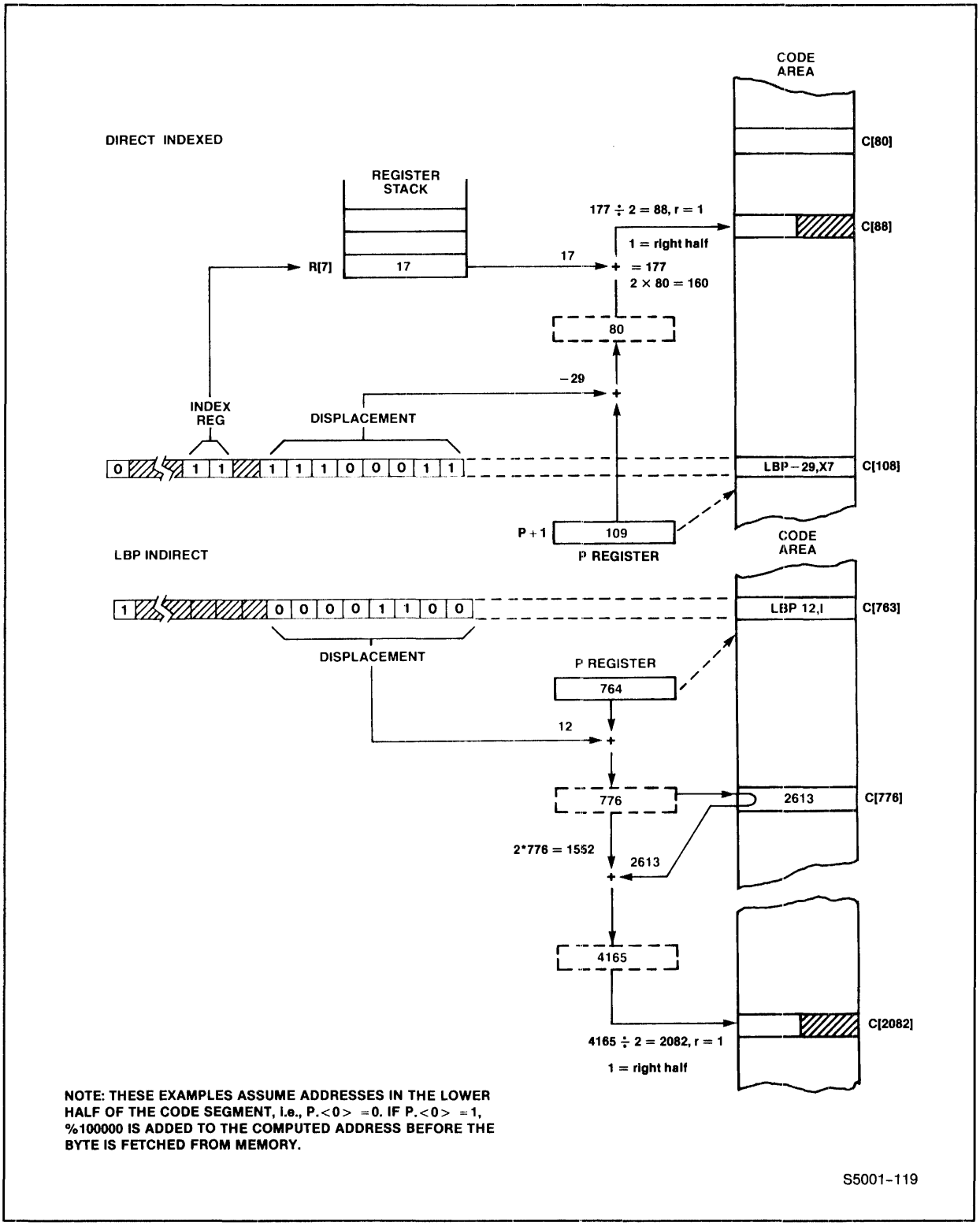


Figure 9-7. LBP Instruction Addressing

LDX (-3----). Load Index Register from Data Space. The index register specified by the "x" field of the instruction is loaded with the contents of the effective memory address. Condition Code is set. Figure 9-8 shows the instruction word format for memory data reference instructions, such as LDX.

NSTO (-34---). Nondestructive Store from A. The contents of the A Register are stored into effective address memory location. The Register Stack is not modified.

LOAD (-40---). Load A from Data Space. The contents of the effective address memory location are pushed onto the stack. Condition Code is set.

STOR (-44---). Store A into Data Space. The contents of the A Register are stored into the effective memory location. A is then deleted from the stack.

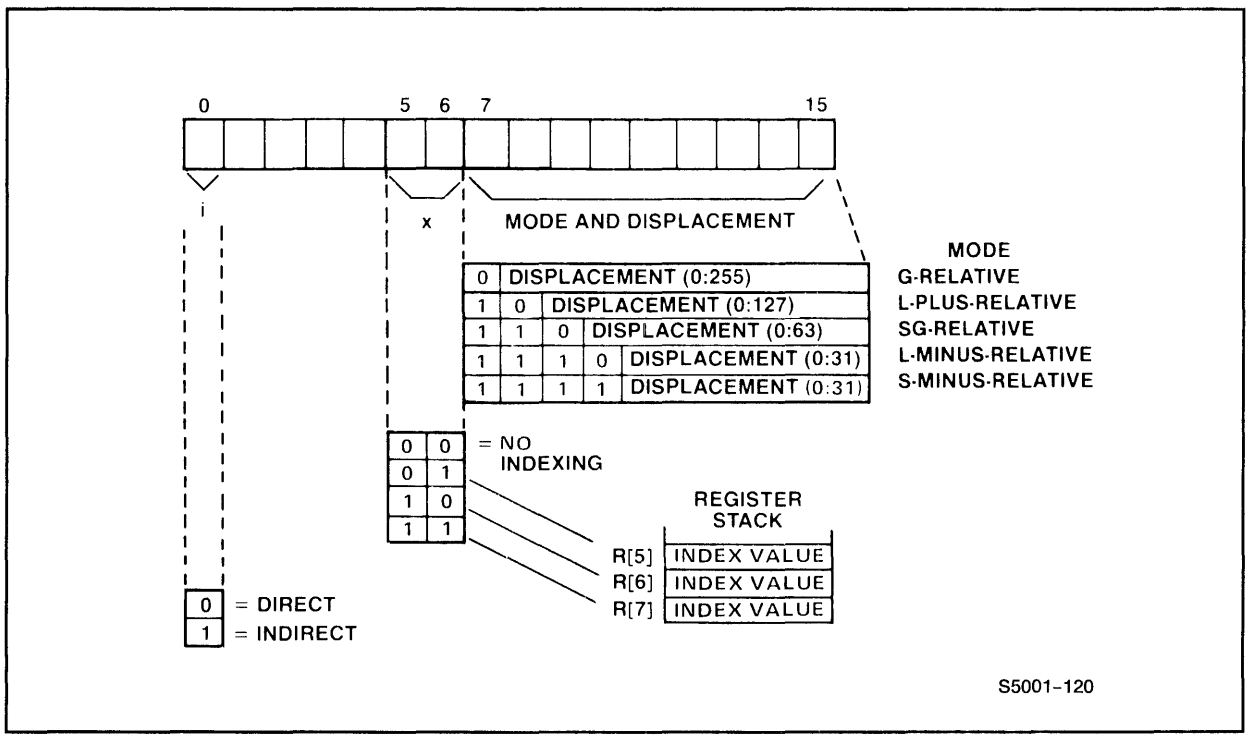


Figure 9-8. Memory Reference Instruction Format

LDB (-5----). Load A with Byte from Data Space. The contents of the effective memory location are loaded into bits 8:15 of A. (Refer to Figure 4-12 in Section 4 for calculation of the effective address in byte addressing.) The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character.

STB (-54---). Store Byte from A to Data Space. The contents of the byte in bits 8:15 of A are stored in the effective memory location. (Refer to Figure 4-12 in Section 4 for calculation of the effective address in byte addressing.)

LDD (-6----). Load Double from Data Space into BA. The doubleword integer contained in the effective memory location is pushed into the stack. Condition Code is set. Figure 9-9 illustrates the addressing methods for doubleword instructions.

STD (-64---). Store Double from BA into Data Space. The contents of BA are stored in the effective memory location. BA is deleted.

LADR (-7----). Load G-Relative Address of Variable into A. The G-relative address of the variable is pushed onto the stack.

ADM (-74---). Add A to Variable in Data Space. The A Register is added in integer form to the contents of the effective memory location and the Condition Code is set on the sum. Overflow is set if the result is greater than 32767 or less than -32768. Carry can also be set. A is then deleted from the stack.

NOTE

For binary coding details of the following six instructions (PUSH through SBXX), refer to Table A-5 in Appendix A.

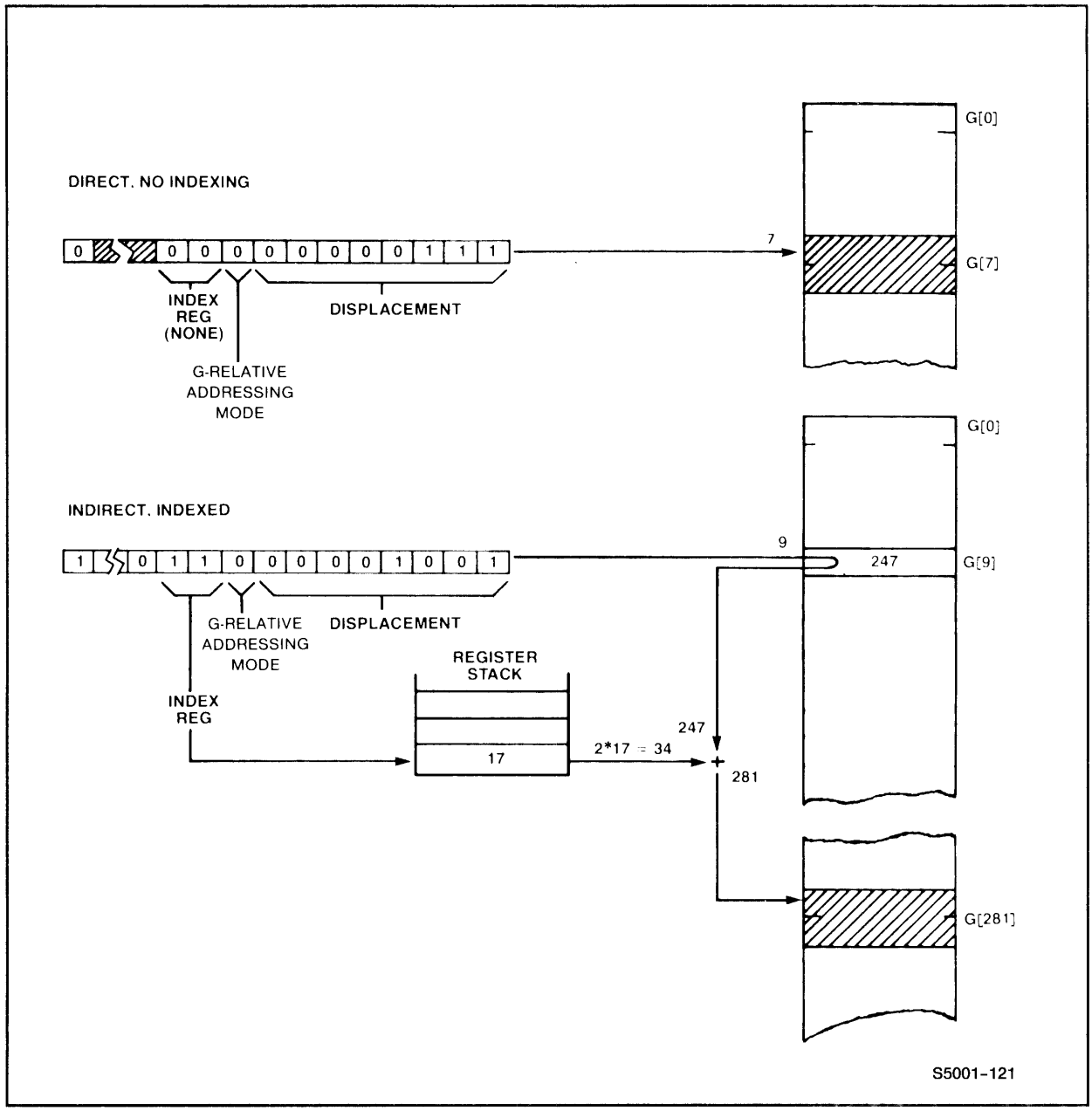


Figure 9-9. Doubleword Addressing

PUSH (024nrc). Push Registers to Data Space. This instruction transfers the contents of a specified number of elements in the Register Stack to the top of the data stack in memory. The "n" field of the instruction is the value to which RP will be set following the instruction; the "r" field specifies the last register stack element to be pushed; the "c" field is the number

of registers minus one that will be pushed to memory. Following the PUSH instruction, the S register points to the last element pushed onto the memory stack. If the resultant value of S is greater than %77777, a stack overflow trap occurs. Figure 9-10 illustrates the bit fields and the action of the PUSH instruction.

POP (124nrc). Pop Data Space to Registers. This instruction loads the Register Stack with the top elements of the data stack (as indicated by the current S register setting). The "n" field of the instruction indicates the value RP will have following the instruction; the "r" field specifies the last Register Stack element to be loaded from memory; the "c" field specifies the number of registers minus one that will be loaded. If the resultant value of S is greater than %77777, a stack overflow trap occurs. Figure 9-10 illustrates the bit fields and the action of the POP instruction.

LWXX (0254--, 0264--). Load Word Extended, Indexed. The word contained in a computed extended memory location is loaded onto the stack, replacing the prior contents of A. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either G[0] (coded 0254--) or the current L register value (coded 0264--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is sign-extended, then arithmetically shifted left one bit position (multiplication by 2, since this instruction requires word addressing rather than byte addressing) and is then added to the extended memory pointer to address the word that is to be loaded. Condition Code is set.

SWXX (0255--, 0265--). Store Word Extended, Indexed. The word contained in B is stored into a computed extended memory location. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either G[0] (coded 0255--) or the current L register value (coded 0265--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is sign-extended, then arithmetically shifted left one bit position (multiplication by 2, since this instruction requires word addressing rather than byte addressing) and is then added to the extended memory pointer to address the location that is to receive the word being stored.

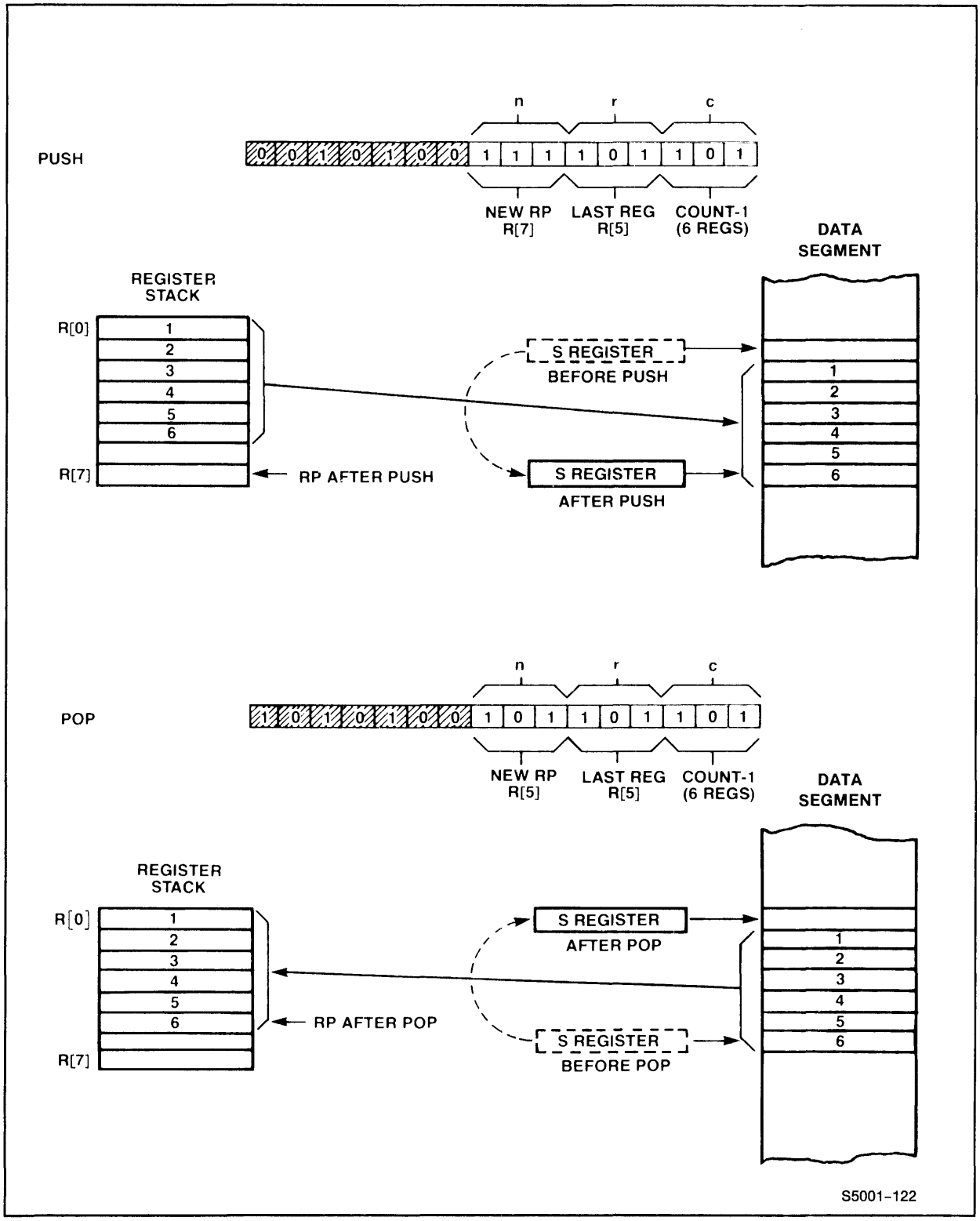


Figure 9-10. PUSH and POP Instructions

LBXX (0256--, 0266--). Load Byte Extended, Indexed. The byte contained in a computed extended memory location is loaded onto the stack, replacing the prior contents of A. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either G[0] (coded 0256--) or the current L register value (coded 0266--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is then added to the extended memory pointer to address the byte that is to be loaded. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

SBXX (0257--, 0267--). Store Byte Extended, Indexed. The byte contained B.<8:15> is stored into a computed extended memory location. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either G[0] (coded 0257--) or the current L register value (coded 0267--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is then added to the extended memory pointer to address the location that is to receive the byte being stored.

LOAD AND STORE BY ADDRESS ON REGISTER STACK

ANS (000034). AND to SG Memory. The word in B is logically ANDed to a word in the system data segment that is specified by a 16-bit address in A. The result remains in the system data location, and A and B are deleted from the stack. If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (system or user) is determined by the DS bit (bit 6) of the ENV register. Condition Code is set.

ORS (000035). OR to SG Memory. The word in B is logically ORed to a word in the system data segment that is specified by a 16-bit address in A. The result remains in the system data location, and A and B are deleted from the stack. If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (system or user) is determined by the DS bit (bit 6) of the ENV register. Condition Code is set.

ANG (000044). AND to Memory. The word in B is logically ANDed to a word in the current data segment that is specified by a 16-bit address in A. The result remains in the data segment location, and A and B are deleted from the stack. Condition Code is set.

ORG (000045). OR to Memory. The word in B is logically ORed to a word in the current data segment that is specified by a 16-bit address in A. The result remains in the data segment location, and A and B are deleted from the stack. Condition Code is set.

ANX (000046). AND to Extended Memory. The word in C is logically ANDed to a word in extended memory that is specified by a 32-bit address in BA. The result remains in the memory location, and A, B, and C are deleted from the stack. Condition Code is set.

ORX (000047). OR to Extended Memory. The word in C is logically ORed to a word in extended memory that is specified by a 32-bit address in BA. The result remains in the memory location, and A, B, and C are deleted from the stack. Condition Code is set.

LWUC (000342). Load Word from User Code Space. A word in the user code segment, specified by a 16-bit address in A, is loaded onto the stack, replacing the prior contents of A. Condition Code is set.

LWAS (000350). Load Word via A from System. The word contained in the effective memory location pointed to by the address in A is loaded onto the stack, replacing the prior contents of A. If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (system or user) is determined by the DS bit (bit 6) of the ENV register. Condition Code is set.

LWA (000360). Load Word via A. The word contained in the effective memory location pointed to by the address in A is loaded onto the stack, replacing the prior contents of A. LWA accesses the current data segment only. Condition Code is set.

SWAS (000351). Store Word via A into System. The word contained in B is stored into the effective memory location pointed to by the address in A. Both words are then deleted from the stack. If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV register.

SWA (000361). Store Word via A. The word contained in B is stored into the effective memory location pointed to by the address in A. Both words are then deleted from the stack. SWA accesses the current data segment only.

LDAS (000352). Load Double via A from System. The doubleword contained in the effective memory locations starting at the location pointed to by the address in A is loaded into BA (after the address in A is deleted). If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV register. Condition Code is set.

LDA (000362). Load Double via A. The doubleword contained in the effective memory locations starting at the location pointed to by the address in A is loaded into BA (after the address in A is deleted). LDA accesses the current data segment only. Condition Code is set.

SDAS (000353). Store Double via A into System. The doubleword in CB is stored into the effective memory locations starting at the location pointed to by the address in A. CBA is then deleted. If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV register.

SDA (000363). Store Double via A. The doubleword in CB is stored into the effective memory locations starting at the location pointed to by the address in A. CBA is then deleted. SDA accesses the current data segment only.

LBAS (000354). Load Byte via A from System. The byte contained in the effective memory location pointed to by the byte address in A is loaded onto the stack, replacing the prior contents of A. If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV register. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

LBA (000364). Load Byte via A. The byte contained in the effective memory location pointed to by the byte address in A is loaded onto the stack, replacing the prior contents of A. LBA accesses the current data segment only. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

SBAS (000355). Store Byte via A into System. The byte in B is stored into the effective memory location pointed to by the byte address in A. Both B and A are then deleted. If privileged mode is in effect when this instruction is executed, A refers to an address in the system data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV register.

SBA (000365). Store Byte via A. The byte in B is stored into the effective memory location pointed to by the byte address in A. Both B and A are then deleted. SBA accesses the current data segment only.

DFS (000357). Deposit Field into System Data. Using the mask bits in register B, this instruction deposits the bits in register C into the location specified by the 16-bit address in A. A, B, and C are then deleted. (See Figure 9-4 and DPF description under "Bit Deposit and Shift" for further details on this operation.) If privileged mode is in effect, the destination is in the system data segment; otherwise, the destination is in the current data segment. A, B, and C are then deleted. Condition Code is set.

DFG (000367). Deposit Field in Memory. Using the mask bits in register B, this instruction deposits the bits in register C into the location specified by the 16-bit address in A. A, B, and C are then deleted. (See Figure 9-4 and DPF description under "Bit Deposit and Shift" for further details on this operation.) DFG accesses the current data segment. Condition Code is set.

LBX (000406). Load Byte Extended. The byte in the extended memory location specified by the 32-bit address in registers B and A is loaded onto the Register Stack (bits 8 through 15 of A), after the address in BA is deleted. The left byte is zero. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

SBX (000407). Store Byte Extended. The byte in bits 8 through 15 of C is stored into the extended memory location specified by the 32-bit address in registers B and A. C, B, and A are then deleted.

LWX (000410). Load Word Extended. The word in the extended memory location specified by the 32-bit address in registers B and A is loaded into register A (after the address in BA is deleted). Condition Code is set.

SWX (000411). Store Word Extended. The word in register C is stored into the extended memory location specified by the 32-bit address in registers B and A. C, B, and A are then deleted.

LDDX (000412). Load Doubleword Extended. The doubleword starting at the extended memory location specified by the 32-bit address in registers B and A is loaded onto the register stack, replacing the prior contents of B and A. Condition Code is set.

SDDX (000413). Store Doubleword Extended. The doubleword in registers D and C is stored into extended memory starting at the location specified by the 32-bit address in registers B and A. All four words are then deleted from the Register Stack.

LQX (000414). Load Quadrupleword Extended. The quadrupleword starting at the extended memory location specified by the 32-bit address in registers B and A is loaded into registers DCBA of the Register Stack (after the address in BA is deleted). Condition Code is set.

SQX (000415). Store Quadrupleword Extended. The quadrupleword in registers FEDC is stored into extended memory (8 bytes) starting at the location specified by the 32-bit address in registers B and A. All six words are then deleted from the Register Stack.

DFX (000416). Deposit Field Extended. Using the mask bits in register C, this instruction deposits the bits in register D into the extended memory location specified by the 32-bit address in registers B and A. All four words are then deleted from the Register Stack. (See Figure 9-4 and DPF description under "Bit Deposit and Shift" for further details on this operation.) Condition Code is set.

SCS (000444). Set Code Segment. Registers B and A are assumed to contain a 17-bit byte address. This instruction sets a logical segment number into the segment number field (bits 0 through 14 of B) to formulate a complete 32-bit address. Only two values may be set for this field: 2 (indicating current code segment) if either the CS or LS bit of the Environment Register contains a one; 3 (indicating user code segment) if both of these bits are zero.

LQAS (000445). Load Quadrupleword via A from SG. The quadrupleword contained in the four memory locations starting at the location pointed to by the address in A is loaded into DCBA (after the address in A is deleted). The address in A refers to an address in the system data segment. Condition Code is set. This is a privileged instruction.

SQAS (000446). Store Quadrupleword via A to SG. The quadrupleword in registers EDCB is stored into the four memory locations starting at the location pointed to by the address in A. The address in A refers to an address in the system data segment. All five words are then deleted from the Register Stack. This is a privileged instruction.

BRANCHING

NOTE

For binary coding details of the following branch instructions, refer to Table A-6 in Appendix A.

BIC (-100--). Branch if CARRY. If the carry bit (K) in the Environment Register is set (K = 1), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed. Figure 9-11 compares direct and indirect branching.

BUN (-104--). Branch Unconditionally. A direct or indirect unconditional branch is taken (depending on the "i" field of the instruction).

BOX (-1-4--). Branch on X Less Than A and Increment X. If the index register as specified by the "x" field of the instruction is less than A, that index register is incremented and a direct or indirect branch is taken (depending on the "i" field of the instruction). If X is greater than or equal to A, A is deleted from the stack and the next instruction is executed.

BGTR (-11---). Branch if CC is Greater. If the Condition Code in the ENV register is CCG (N = 0, Z = 0), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

BEQL (-12---). Branch if CC is Equal. If the Condition Code in the ENV register is CCE (N = 0, Z = 1), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

BGEQ (-13---). Branch if CC is Greater or Equal. If the Condition Code in the ENV register is CCG or CCE (N = 0) a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

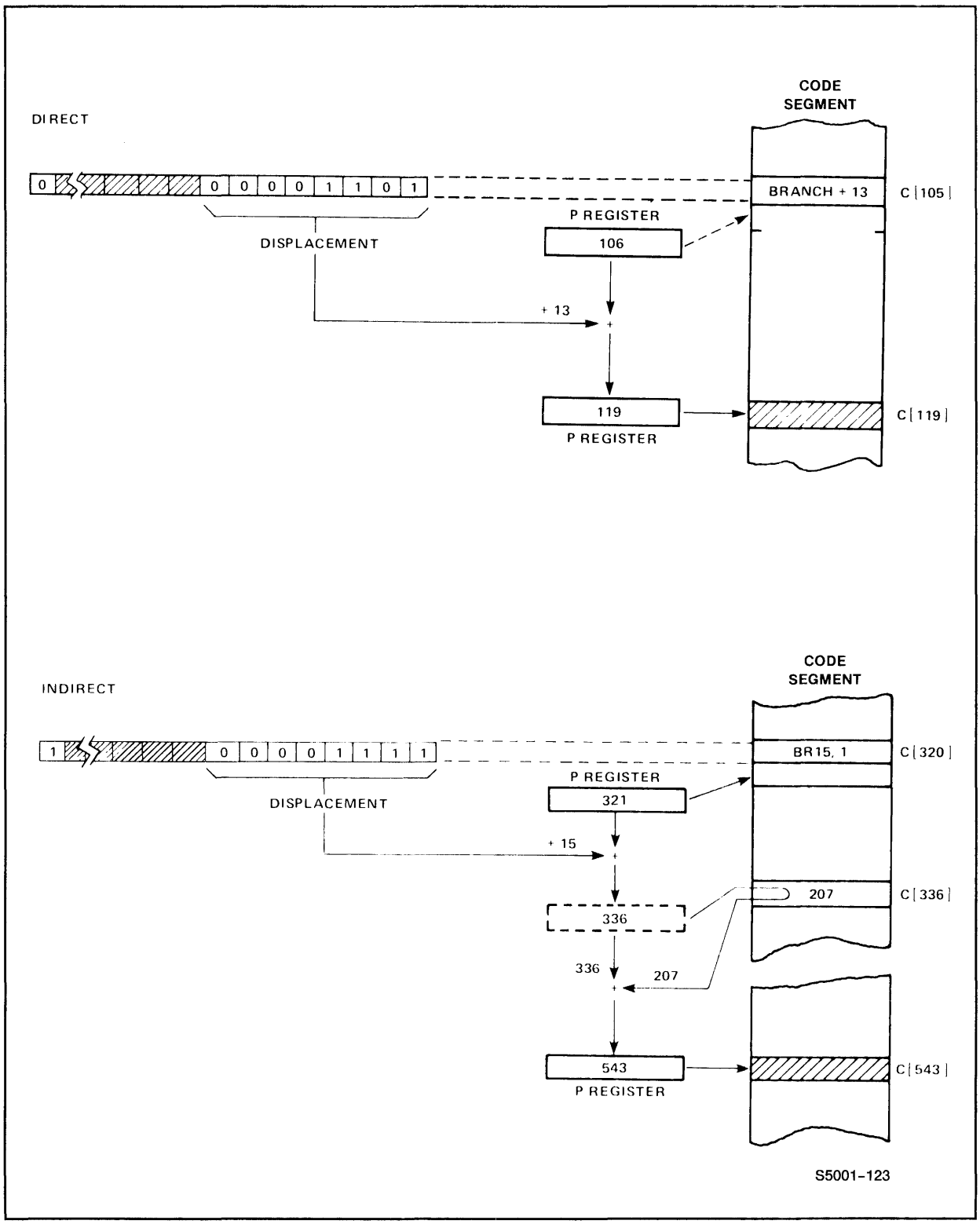


Figure 9-11. Direct vs. Indirect Branching

BLSS (-14---). Branch if CC is Less. If the Condition Code in the ENV register is CCL (N = 1), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

BAZ (-144--). Branch on A Zero. If the A Register equals zero, a direct or indirect branch is taken (depending on the "i" field of the instruction). If the A Register does not equal zero, the next instruction is executed. In either case, A is deleted from the stack.

BNEQ (-15---). Branch if CC is not equal. If the Condition Code in the ENV register is not CCE (Z = 0), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

BANZ (-154--). Branch on A Not Zero. If the A Register is non-zero, a direct or indirect branch is taken (depending on the "i" field of the instruction). If the A Register equals zero, the next instruction is executed. In either case, A is deleted from the stack.

BLEQ (-16---). Branch if CC is Less or Equal. If the Condition Code in the ENV register is CCL or CCE (N = 1 or Z = 1), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

BNOV (-164--). Branch if no OVERFLOW. If the Overflow bit (V) in the ENV register is not set (V = 0), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

BNOC (-17---). Branch if no CARRY. If the Carry bit (K) in the ENV register is not set (K = 0), a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, the next instruction is executed.

BFI (000030). Branch Forward Indirect. The instruction expects an offset from the current P register setting to be contained in A. An indirect branch is then made through the location specified by P + A. Figure 9-12 illustrates the action of the BFI instruction.

MOVES, COMPARES, SCANS, AND CHECKSUM COMPUTATIONS

MNGG (000226). Move Words While Not Duplicate. Register D is assumed to contain a destination address in the current data segment, and register C is assumed to contain a source address in the current data segment. The MNGG instruction moves words from the source to the destination while the count value in register B is not zero and the source word is not equal to the word in A. The word in A is always the previous word moved. The instruction stops on the first duplicate word or on zero count. After execution, the word in A is deleted, so that A then contains the count, B contains the source address, and C contains the destination address. Interrupts can occur after each word moved.

CDG (000366). Count Duplicate Words. Beginning at the address (in the current data segment) specified in register C, and for a maximum count of words specified in register B, this instruction counts the number of duplicate words in the buffer. Register A is incremented on each duplicate found, and may contain an initial value. After execution, A contains the original A value plus the number of duplicate words, B contains a count of the words left in the buffer (zero if empty), and C contains the address of the first word that did not match its predecessor (or the word after the last word in the buffer). The comparison actually starts with the words specified by C and C-1. This instruction is intended to be used in conjunction with MNGG. Interrupts can occur after each compare.

NOTE

For binary coding details of the following six move instructions (MOVW, MOVB, COMW, COMB, SBW, SBU), refer to Table A-5 in Appendix A. Also, for these six instructions, it is possible to specify either ascending or descending directions. Figure 9-13 provides a comparison of ascending and descending moves, compares, and scans, as described in the following paragraphs. Bit 9 of the instruction word specifies ascending (0) or descending (1).

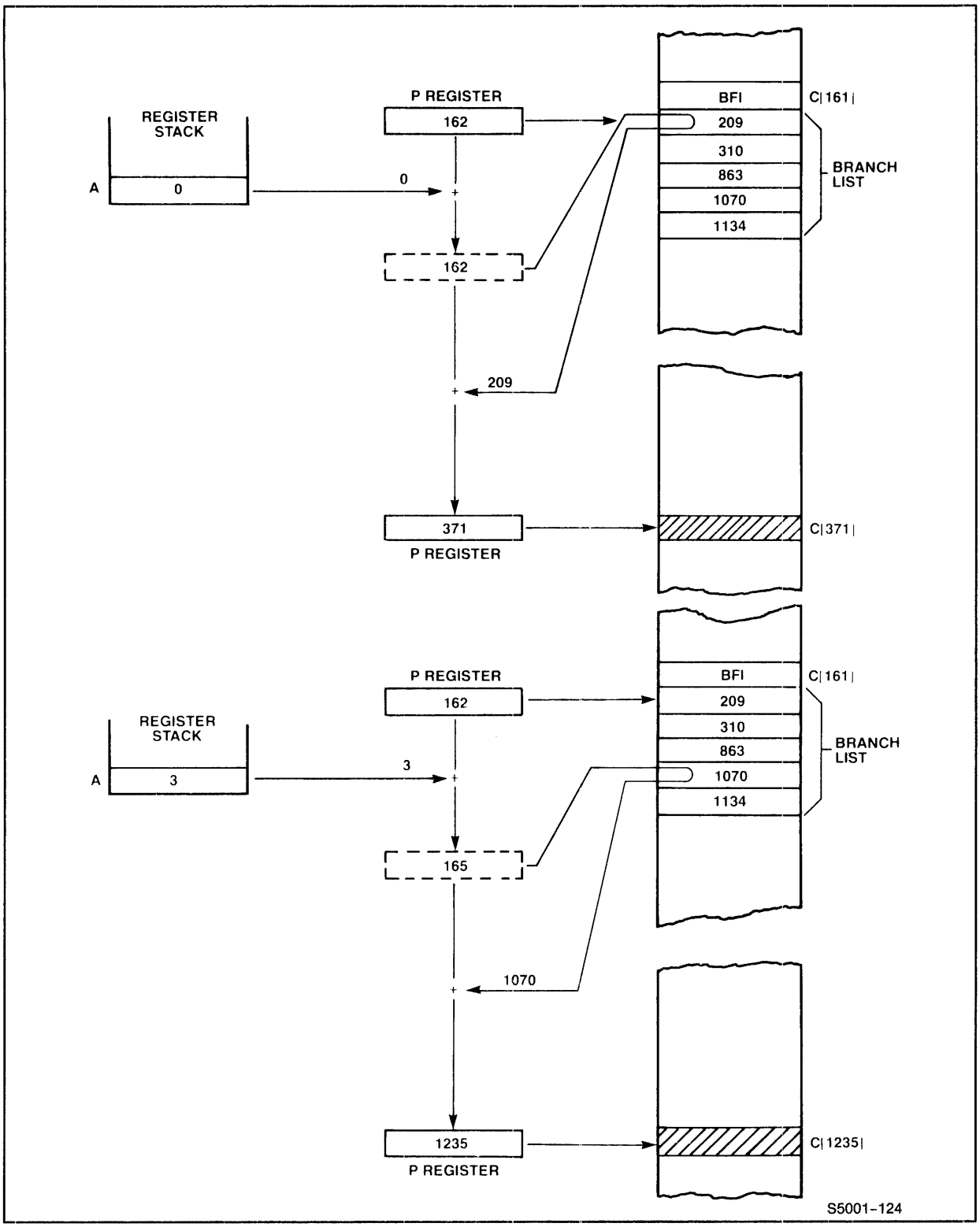


Figure 9-12. Branch Forward Indirect

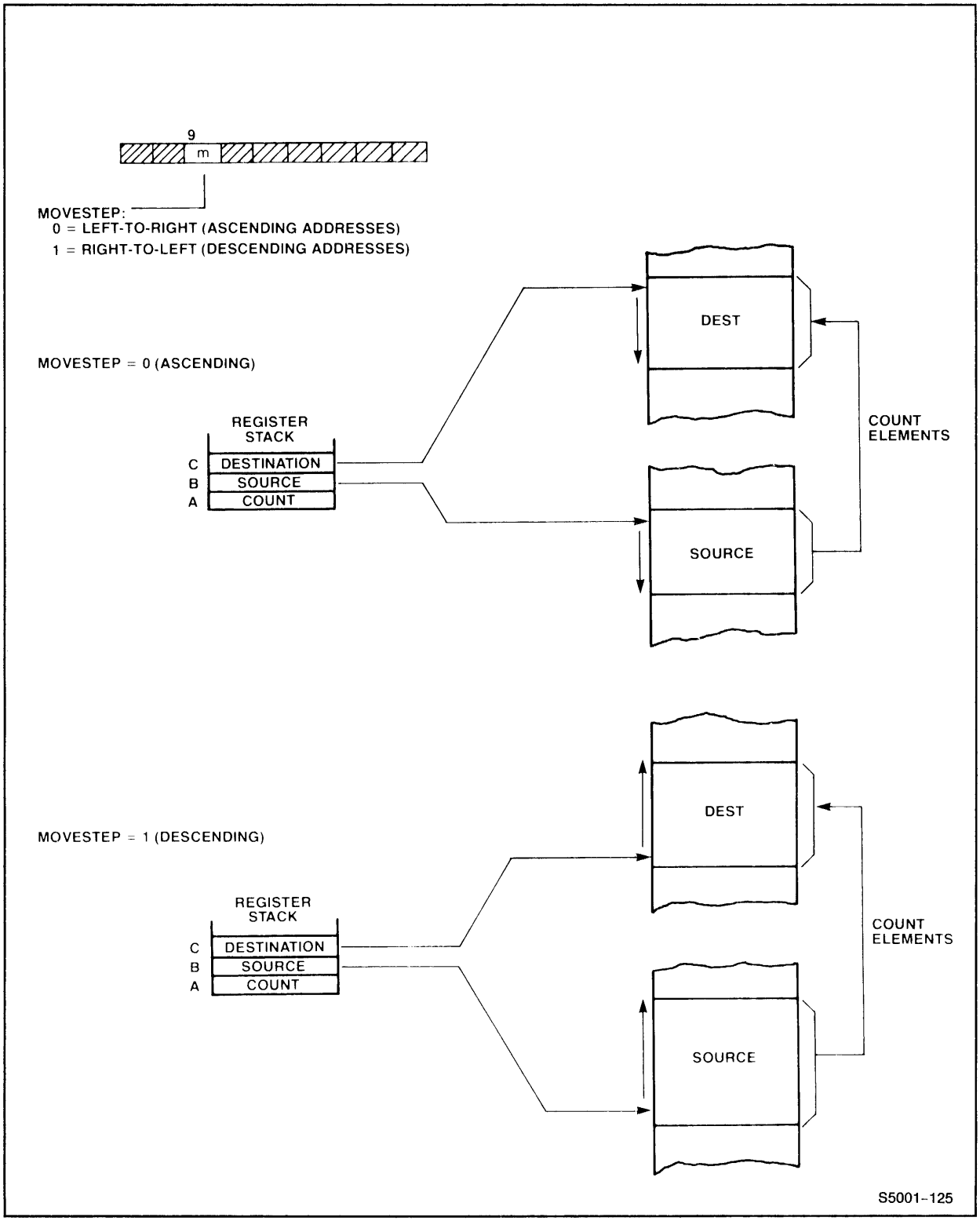


Figure 9-13. Direction for Moves, Compares, and Scans

MOVW (026---). Move Words. This instruction transfers a specified number of words from one area of memory to another. The instruction expects A to contain a word count, B to contain the source word address, and C to contain the destination word address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and PRIV bits of the ENV register. The "m" field of the instruction (see format diagram at the top of Figure 9-13) determines whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each move. The "n" field of the instruction is the value to which RP is set upon instruction end. The move is made one word at a time from the source to the destination. After each word transfer the addresses are decremented or incremented and A is decremented. If A is equal to zero the instruction ends; otherwise the next word is moved. Interrupts can occur after each word moved.

MOVB (126---). Move Bytes. This instruction transfers a specified number of bytes from one area of memory to another. The instruction expects A to contain a byte count, B to contain the source byte address, and C to contain the destination byte address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and PRIV bits of the ENV register. The "m" field of the instruction determines whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each move. The "n" field of the instruction is the value to which RP is set upon instruction end. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are decremented or incremented and A is decremented. If A is equal to zero, the instruction ends; otherwise the next byte is moved. If the source is a code segment and the P register currently indicates an address in the upper half of the code segment (bit 0 of P = 1), %100000 is added to the computed address, so that the source and destination addresses will always be relative to whichever half of the segment P currently indicates. Interrupts can occur after each destination word (two bytes) moved.

COMW (0262--). Compare Words. This instruction compares one area of memory with another, a word at a time, until a miscomparison occurs or until a specified number of comparisons have been made. The words being compared are treated as unsigned quantities. COMW expects A to contain a word count, B to contain a source word address and C to contain a destination word address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and PRIV bits of the ENV register. The "m" field determines

whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each comparison. The "n" field is the value to which RP will be set upon instruction termination. The instruction fetches the contents of source and destination addresses, compares them, increments or decrements the address by one according to the "m" field, and decrements the word count in A until either A = 0 or a miscomparison is reached. If termination is due to a miscomparison, CC indicates the results of the compare or CCE due to A going to zero. Interrupts can occur after each comparison.

COMB (1262--). Compare Bytes. This instruction compares one area of memory with another, a byte at a time, until the bytes are not equal or until a specified number of comparisons have been made. It expects A to contain a byte count, B to contain a source byte address and C to contain a destination byte address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and PRIV bits of the ENV register. If the source address is in a code segment, the byte address is taken to be in the same 64K half of the code space as the current P register value. The "m" field determines whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each comparison. The "n" field is the value to which RP will be set upon instruction termination. The instruction fetches the contents of source and destination addresses, compares them, increments or decrements the address by one according to the "m" field, and decrements the byte count in A until either A = 0 or a miscomparison is reached. If termination is due to a miscomparison, CCG indicates that the byte at C is greater than the byte at B, or CCL indicates that the byte at C is less than the byte at B; A indicates the number of bytes left to compare. If termination is due to the count running out, CCE indicates that all bytes compared exactly, and C and B will point to the next locations not compared. Interrupts can occur after each comparison.

SBW (1264--). Scan Bytes While. The SBW instruction expects A to contain a comparison byte in bits 8:15 and B to contain the byte address of the string to be scanned. The map to be used is determined by the "s" field of the instruction and by the DS, CS, LS, and PRIV bits of the ENV register. The "m" field of the instruction determines whether the source address will be incremented ("m" = 0) or decremented ("m" = 1) after each comparison. The scan is terminated when either a null byte is found in the string or a byte in the string does not match the test byte in A. When null byte termination occurs, the Carry (K) bit in the ENV Register is set. In either termination case, B

points to the byte address that caused termination. RP is set to the "n" field of the instruction at instruction termination. Interrupts can occur after each comparison.

SBU (1266--). Scan Bytes Until. The SBU instruction expects A.<8:15> to contain a test byte and B to contain the byte address of the string to be scanned. The map to be used is determined by the "s" field of the instruction and by the DS, CS, LS, and PRIV bits of the ENV register. The "m" field of the instruction determines whether the scan address will be incremented ("m" = 0) or decremented ("m" = 1) after each comparison. The scan is terminated when either a null byte is found in the string or the test byte matches a byte in the string. The Carry (K) bit is set in the ENV register when null byte termination occurs. In either case, B points to the byte address that caused the scan to cease. RP is set to the "n" field of the instruction at termination. Interrupts can occur after each comparison.

MNDX (000227). Move Words While Not Duplicate, Extended. FE is assumed to contain a 32-bit destination address in extended memory, and DC is assumed to contain a 32-bit source address. The MNDX instruction moves words from the source to the destination while the count value in register B is not zero and the source word is not equal to the word in A. The word in A is always the previous word moved. The instruction stops on the first duplicate word or on zero count. After execution, the word in A is deleted, so that A then contains the count, CB contains the source address, and ED contains the destination address. Interrupts can occur after each word has been transferred.

CDX (000356). Count Duplicate Words, Extended. Beginning at the 32-bit address (in extended memory) specified in DC, and for a maximum count of words specified in B, this instruction counts the number of duplicate words in the buffer. A is incremented on each duplicate found, and may contain an initial value. After execution, A contains the original A value plus the number of duplicate words, B contains a count of the words left in the buffer (zero if empty), and DC contains the extended address of the first word that did not match its predecessor (or the word after the last word in the buffer). The comparison actually starts with the words specified by DC and DC-2. Interrupts can occur after each comparison. This instruction is intended to be used in conjunction with MNDX.

MVBX (000417). Move Bytes Extended. This instruction transfers a specified number of bytes from one area of extended memory to another. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, and ED to contain a 32-bit destination byte address. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are incremented and A is decremented. If A is equal to zero the instruction ends; otherwise the next byte is moved. All five words are deleted from the stack when the instruction ends. Interrupts can occur after each byte has been transferred.

MBXR (000420). Move Bytes Extended, Reverse. This instruction transfers a specified number of bytes from one area of extended memory to another, using reverse (decrementing) addresses. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, and ED to contain a 32-bit destination byte address. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are decremented and A is decremented. If A is equal to zero the instruction ends; otherwise the next byte is moved. All five words are deleted from the stack when the instruction ends. Interrupts can occur after each byte transferred.

MBXX (000421). Move Bytes Extended, and Checksum. This instruction transfers a specified number of bytes from one area of extended memory to another, and computes a checksum value (byte exclusive "or") after each byte is moved. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, ED to contain a 32-bit destination byte address, and F to contain the initial checksum value. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are incremented, A is decremented, and new checksum is entered in F. If A is equal to zero, the instruction ends; otherwise the next byte is moved. Five words are deleted from the Register Stack when the instruction ends, leaving the final checksum value in A. Interrupts can occur after each byte has been transferred.

CMBX (000422). Compare Bytes Extended. This instruction compares one area of extended memory with another, a byte at a time, until the bytes are not equal or until a specified number of comparisons have been made. Before beginning the compare, CMBX checks to make sure that both strings in the compare are mapped in extended memory. This instruction expects A to contain a byte count, CB to contain a 32-bit source byte address and ED

to contain a 32-bit destination byte address. The instruction fetches the contents of the source and destination addresses, compares them, increments the addresses by one, and decrements the byte count in A until either A = 0 or a noncomparison is reached. If termination is due to a noncomparison, CCG indicates that the byte at ED is greater than the byte at CB, or CCL indicates that the byte at ED is less than the byte at CB; A indicates the count of bytes left to compare. If termination is due to the count running out, CCE indicates that all bytes compared exactly; ED and CB point to the bytes after the last ones compared, and A is 0. Interrupts can occur after each comparison.

XSMG (000343). Compute Checksum in Current Data. Starting at the address defined in register B, for a count of words defined in register A, the XSMG instruction exclusive-ORs each word into register C. When the count goes to zero, the two top words on the stack are deleted, leaving the final checksum in register A. The address in B refers to the current data segment only. Interrupts can occur after each word checksummed.

XSMX (000333). Compute Checksum Extended. Starting at the extended memory location defined by the 32-bit address in CB, for a count of words defined in register A, the XSMX instruction exclusive-ORs each word into register D. When the count goes to zero, the three top words on the stack are deleted, leaving the final checksum in register A. Interrupts can occur after each word checksummed.

PROGRAM REGISTER CONTROL

SETL (000020). Set L with A. The contents of the L register, which points to the current stack marker, are replaced with the contents of register A. A is then deleted from the Register Stack.

SETS (000021). Set S with A. The contents of the S register, which points to the top word of the stack in memory, are replaced with the contents of register A. A is then deleted from the stack. A stack overflow trap occurs if the result is greater than 32767.

SETE (000022). Set ENV with A. The least significant eight bits of the Environment Register (ENV) are replaced with the lower eight bits of the A Register. The most significant eight bits of the Environment Register are logically ANDed with the upper eight bits of the A Register. Thus this instruction may only clear the PRIV, DS, CS, and LS bits of the Environment Register, and may not set them. The programmer should take care with this instruction on NonStop II systems, since it is possible to inadvertently clear the Library Space (LS) bit, ENV.<4>.

SETP (000023). Set P with A. The contents of the Program Counter (P) are replaced with the contents of the A Register. A is deleted from the stack, and control is transferred to the new location indicated by P.

RDE (000024). Read ENV into A. The contents of the Environment Register (ENV) are pushed onto the Register Stack.

RDP (000025). Read P into A. The contents of the Program Counter (P) are pushed onto the Register Stack.

STRP (00010-). Set RP. The register pointer is set to the value in the Register field of the instruction. For binary coding details, see Table A-7 in Appendix A.

ADDS (002---). Add Immediate Operand to S. The signed immediate operand is added to the S register in integer form. If the resultant S is greater than 32767, then a stack overflow trap occurs.

CCL (000015). Set Condition Code to Less. A Condition Code of CCL (N = 1 and Z = 0) is set into the ENV register.

CCE (000016). Set Condition Code to Equal. A Condition Code of CCE (N = 0 and Z = 1) is set into the ENV register.

CCG (000017). Set Condition Code to Greater. A Condition Code of CCG (N = 0 and Z = 0) is set into the ENV register.

ROUTINE CALLS AND RETURNS

PCAL (027---). Procedure Call. Control is transferred to an instruction specified by an entry in the Procedure Entry Point (PEP) Table; the specific PEP entry is indicated by the PEP Number field of the instruction. First, a three-word stack marker, consisting of the current P, ENV, and L, is stored on the top of the current stack. (ENV includes the space ID index in bits 11:15; CC and RP are not preserved.) If the caller is not privileged, the PEP number is checked against PEP[0] and PEP[1] to see if the call is legal. If the call is not legal, an instruction failure trap occurs. (If the caller is privileged no checks are made.) L and S are set to S + 3 to point to the base of a new local data area. The final value of S is then checked for a value greater than 32767; if it is, a stack overflow trap occurs. Finally, P is set from the PEP entry and control is transferred to the procedure.

XCAL (127---). External Procedure Call. The XCAL instruction is used to invoke procedures that are outside the current code segment. Like PCAL, XCAL creates a three-word stack marker. Then control is transferred to an instruction in the external segment by a three-step sequence: 1) a number in the XEP field of the instruction refers to an entry in the XEP table of the current code segment; 2) the XEP entry specifies a segment and a PEP entry in that segment; 3) the PEP entry of the other code segment specifies a procedure entry point within that segment. See detailed description in Section 4 under the heading, "Calling External Procedures".

SCMP (000454). Set Code Map. This instruction is used to establish a procedure label in register A for use by the DPCL instruction (described next). If the label to be passed is for a procedure in the current code segment (signified by A.<0:6> = 0), the PEP index is expected to be in A.<7:15>, and SCMP will insert the space ID of the current code segment in A.<0:6>, thus forming a complete procedure label. If the label to be passed is for a procedure in some other code segment (as indicated by A.<0:6>=%133), the XEP index is expected to be in A.<7:15>, and SCMP will load that XEP entry (which is already in procedure label format) into A.<0:15>. In typical usage, succeeding instructions would pass this value to a procedure which would then issue the DPCL instruction.

DPCL (000032). Dynamic Procedure Call. Control is transferred to a procedure which is dynamically specifiable in the Register Stack (Register A). The specified procedure may be in any of the four short address code spaces (UC, SC, UL, SL). The format of the word in Register A for specifying the target procedure is the same as that for a XEP table entry (see Figure 4-26). DPCL first stores a three-word stack marker, consisting of the current P, ENV, and L, on the top of the stack. (ENV includes the caller's space ID index in bits 11:15.) Then a check is made to see if the target segment is currently mapped; if not, a MAPS (Map Segment) instruction is executed at this point. Then, if the caller is not privileged, the PEP number is checked to see if the call is legal. If the call is not legal, an instruction failure trap occurs. If the caller is privileged, this check is not made. L and S are set to S + 3 to point to the base of a new local data area. The final value of S is then checked for a value greater than 32767; if it is, a stack overflow trap occurs. Next, if the call is to a callable system procedure, the PRIV bit in the ENV Register is set. CS and LS of ENV are set according to the corresponding bits of A (0 and 1 respectively). Finally, P is set from the PEP entry, transferring control to the target procedure.

EXIT (125---). Exit from Procedure. This instruction is used to return from a procedure called by a PCAL, XCAL, or DPCL instruction. EXIT assumes L-2:L to contain a standard three-word stack marker consisting of P, ENV, and L. (ENV includes the caller's space ID index in bits 11:15.) The first action of EXIT is to check if the procedure being returned to is currently mapped; if not, a MAPS (Map Segment) instruction is executed at this point to map the return segment. Then S is moved below the current stack marker and any parameters by setting it with the "S decrement" value subtracted from the current L register setting. P is set to the return P value contained in L[-2] of the current stack marker. The caller's ENV register value is set as follows: the mode (privileged or nonprivileged) and data area are reinstated to the lesser of the caller's and the current settings (e.g., a privileged calling process can be made nonprivileged on the return, but not vice versa); the calling process's CS (code space), LS (library space), T (traps), V (overflow), and K (carry) are reinstated from L[-1]; Z and N (Condition Code) and RP are set to those of the current procedure. L is moved back to the preceding stack marker, thereby reinstating the preceding local data area, by setting L with the contents of the L[0] of the current stack marker.

DXIT (000072). DEBUG Exit. This instruction is used to reestablish the environment present at the time DEBUG was called. P, ENV, and L are restored from the stack marker generated by the DEBUG call, and S is reset to its value at the time of the call to DEBUG. Lastly, the instruction checks CSSEG to see if the segment specified by space ID in L-5 is currently mapped. If the segment is not currently mapped, a MAPS (Map Segment) instruction is executed at this point. This is a privileged instruction.

BSUB (-174--). Branch to Subprocedure. S is incremented by one and the return address (P) is saved in that location. Then a direct or indirect unconditional branch is taken (depending on the "i" field of the instruction). For binary coding details, see Table A-6 in Appendix A.

RSUB (025---). Return from Subroutine. This instruction is used to return from a subroutine called by a BSUB instruction. The instruction assumes that the return address is on the top of the memory stack (indicated by S) and returns control to that address. S is set to S - S^decrement. "S^decrement" may be any number from 0 to 255; however, in order to delete the return address from the stack, it must be at least 1. For binary coding details, see Table A-5 in Appendix A.

INTERRUPT SYSTEM

RIR (000063). Reset Interrupt Register. This instruction is used by the operating system interrupt handlers to reset the appropriate INTA Register bit after an interrupt has occurred. Some interrupt bits must be reset (along with the clearing of a MASK bit) in order to allow further interrupts through that SIV (System Interrupt Vector Table) entry. The instruction expects A to contain the number of the bit in the INTA Register that is to be reset. This is a privileged instruction.

XMSK (000064). Exchange MASK with A. The contents of the MASK Register are interchanged with the contents of the A Register. This is a privileged instruction.

IXIT (000071). Interrupt Exit. This instruction is used by the operating system interrupt procedures to return control to the

interrupted process. At the time the interrupt occurred, a stack marker was generated at the L pointed to by the System Interrupt Vector Table (SIV) for the specific interrupt. This was a special six-word marker that consisted of the space ID, MASK, S, P, ENV, and L at the time of the interrupt. This instruction reestablishes this environment and resumes execution of the interrupted process. In order to reestablish the interrupted environment, IXIT first loads the five registers with the values in L-4:L of the stack marker, and then checks CSSEG to see if the segment specified by the space ID in L-5 is currently mapped. If the segment is not currently mapped, a MAPS (Map Segment) instruction is executed at this point. Then the Register Stack is loaded with the values in L+1 through L+8. Lastly, the process timer is allowed to resume counting if the return is to a user environment (DS = 0). At the time this instruction is executed, the needed values in L-5 through L+8 must be present. This is a privileged instruction.

DISP (000073). Dispatch. This instruction sets bit 15 of INTA, and also sets Vi.<15> in the System Interrupt Vector (SIV) table entry for the Dispatcher interrupt. If bit 15 of MASK is set, a Dispatcher interrupt occurs immediately following this instruction (provided there are no interrupts of higher priority pending). Control is then transferred to the operating system Dispatcher whose location is pointed to by the SIV table entry. This is a privileged instruction.

BUS COMMUNICATION

TOTQ (000056). Test Out Queues. In a NonStop II processor this instruction sets CCE if neither of the two Out Queues is full, or CCG if at least one Out Queue is full. In a NonStop TXP processor this instruction sets CCG if the single OUTQ is full, and CCE if empty.

SEND (000065). Send Data over Interprocessor Bus. The SEND instruction expects register A to contain a byte count and registers CB to contain the absolute extended address of the source buffer. Register D is the OUTQ Full Timer.

In a NonStop II processor, the timeout value is computed as: (32768 - <timeout>) times 0.8--this value specifies the time in microseconds for the specified bus to become ready (e.g., <timeout> of 0 = 32768 * 0.8 microseconds).

In a NonStop TXP processor, the timeout value is computed as: (32768 - <timeout>) times 0.833--this value specifies the time in microseconds for the specified bus to become ready (e.g., <timeout> of 0 = 32768 * 0.833 microseconds).

Register E bits 0:7 specify the sender CPU and 8:15 specify the destination CPU. Register F specifies a sequence number, and register G bit 15 specifies which bus is to be used (0 = X, 1 = Y).

Data in the buffer is transmitted in 16-word packets consisting of 26 data bytes (13 words) plus three words for sequence number, sender and receiver CPU numbers, and checksum. Packets are transmitted until the byte count is zero. If the byte count is not a multiple of 26, then the last packet is padded with zeros to round the number of data bytes up to 26. Condition Code CCE indicates successful completion, and the Register Stack is marked empty.

If a timeout condition occurs, a Condition Code of CCL is returned, and the instruction terminates. The Out Queue is cleared. SEND is a privileged instruction.

INPUT-OUTPUT

RSW (000026). Read the Switch Register into A. The contents of the Switch Register are pushed onto the Register Stack. Condition Code is set.

SSW (000027). Store A into Switch Register. The contents of the A Register are set in the Register Display and into sysstack[%122]. A is then deleted.

EIO (000060). Execute Input-Output. The EIO instruction expects bits 8:15 of A to contain the subchannel number, bits 0:7 of A to contain a command to its controller, and 0:15 of B to contain a parameter which is to be passed to that controller via the channel. (In a NonStop TXP processor, before issuing the EIO, you must execute an LIOC to load the IOC entry for a given subchannel into its IOC cache entry.) The EIO instruction first checks to see if the channel is available. If not it loops, waiting for channel availability but testing for other interrupts. When the channel becomes available, the command and address are sent to the controller by the channel via the LAC (Load Address and Command) T-bus command and the parameter is

sent to the controller which is now selected via the LPRM (Load Parameter) T-bus command. Device status is then read from the controller via the RDST (Read Device Status) T-bus command. RP is decremented by one, and if there were no channel errors, device status is placed in A, the controller is then deselected via the DSEL (Deselect) T-bus command, the Condition Code is set to CCE and the instruction terminates. If there was a channel error, the ABTI (Abort Instruction) T-bus command is issued to the controller, deselecting it and terminating its activity. The contents of IOD, although probably invalid due to the channel error, are placed in A for evaluation. The Condition Code is set to CCL and the instruction terminates. This is a privileged instruction.

IIO (000061). Interrogate I/O. This instruction is used by the operating system interrupt handler to get the interrupt cause and interrupt status from a controller and to reset that interrupt. It first checks to see if the channel is available. If not it loops, waiting for channel availability but testing for other interrupts. When the channel is available, first rank 0 and then rank 1 of the I/O system are polled via the LPOL (Low Poll) T-bus command. The interrupting controller on the highest rank with the highest priority is then selected via the SEL (Select) T-bus command. The channel then loads the controller's interrupt cause into the C register, the interrupt status into the B register, and the channel status into the A register. Then the interrupt in the controller is cleared. If there were no channel errors indicated in A, and if interrupt status bits 0:3 are equal to zero, then CCE is set, and the instruction terminates. If there was a channel error then CCL is set, and the instruction terminates. CCG is set in the event of a device error or parity error. This is a privileged instruction.

HIIO (000062). High-Priority Interrogate I/O. This instruction is used by the operating system's high-priority interrupt handler to get the interrupt cause and status from a high-priority controller and to reset the corresponding interrupt. Execution is identical to the IIO instruction, except that HPOL (high priority polls) TBUS commands are issued and only controllers with the high-priority interrupt jumper installed can respond. This is a privileged instruction.

RCHN (000447). Reset I/O Channel. This instruction is used by the operating system to control the I/O channel in the event of a catastrophic error. If register A contains a value greater than or equal to zero, RCHN resets the I/O channel; if A contains a

negative value, RCHN performs a lockup on the channel. Condition Code CCE indicates that the reset or lockup was performed, or CCL indicates that the channel was not available. This is a privileged instruction.

LIOC (000457). Load IOC entry. During an I/O operation, the NonStop TXP processor uses a cached copy of the IOC entry associated with a given subchannel. This technique allows the system to defer updating the memory-resident IOC entry until after the I/O has completed. The LIOC instruction copies the four-word IOC entry from memory to its associated scratchpad registers. The subchannel number is specified in the contents of the A register. This is a privileged instruction. (In a NonStop II processor, this instruction executes as a NOP.)

SIOC (000460). Store IOC entry. In a NonStop TXP processor, this instruction copies the four-word IOC information from scratchpad registers to its associated memory-resident IOC entry. The subchannel number is specified in the A register contents. This is a privileged instruction. (In a NonStop II processor, SIOC executes as a NOP.)

MISCELLANEOUS

NOP (000000). No Operation.

RCLK (000050). Read Clock. This instruction reads the quadrupleword microsecond counter (located in the system data segment), adds the instantaneous value of the 14-bit hardware microsecond counter to it, and pushes the result onto the Register Stack. Note that since the software counter is updated only every 10 milliseconds (each time the hardware counter rolls over), adding the hardware count to it provides an accurate clock indication at the instant that RCLK is executed.

RCPU (000051). Read CPU Number. This instruction reads this processor's CPU number from bits 0:7 of INTB and pushes this value onto the register stack.

BPT (000451). Instruction Breakpoint Trap. This instruction, although necessarily nonprivileged, can be used only by system software (DEBUG); proper operation requires access to the Environment Register, which requires privileged capability. The instruction assumes that DEBUG has inserted the BPT instruction at some user-specified point in the code, and has saved the instruction that formerly occupied that location in the Breakpoint Table in the system data segment. When the code containing the BPT instruction is executed, BPT is normally executed twice--once when encountered following the preceding instruction, and once again to resume program execution at the following instruction. A bit (1) in the Environment Register is used as a flag to differentiate the two functions.

When BPT is first executed, bit 1 of the Environment Register is zero, which causes an interrupt to be generated (through SIV 19) to DEBUG. DEBUG sets ENV bit 1 to one and, after user debugging has been completed, returns to the interrupted code at the BPT instruction. This time, BPT first sets ENV bit 1 back to zero, then searches the Breakpoint Table, locates the saved instruction, loads that instruction into the Instruction (I) Register, and sets the microcode entry point for that instruction into the ROMA Register. Thus the breakpointed instruction is executed, and execution proceeds normally to the succeeding instruction.

OPERATING SYSTEM FUNCTIONS

The following groups of instructions, most of them privileged, are used solely to implement certain operating system and diagnostic functions in firmware. These instructions are not intended for use in any user applications, and are listed here only for completeness.

Resource Management

XCTR (000033) XRAY Counter Bump
MXON (000040) Mutual Exclusion On
MXFF (000041) Mutual Exclusion Off
SNDQ (000052) Signal a Send Is Queued
SFRZ (000053) System Freeze
DOFS (000057) Disc Record Offset
DLEN (000070) Disc Record Length
HALT (000074) Processor Halt
PSEM (000076) "P" a Semaphore
VSEM (000077) "V" a Semaphore
RPV (000216) Read PROM Version Numbers (NonStop II)
WWCS (000400) Write LCS

VWCS	(000401)	Verify LCS
RWCS	(000402)	Read LCS
FRST	(000405)	Firmware Reset
RSMT	(000436)	Read from Operations and Service Processor (OSP)
WSMT	(000437)	Write to Operations and Service Processor (OSP)
RIBA	(000440)	Read INTB and INTA Registers
RPT	(000442)	Read Process Time
SPT	(000443)	Set Process Timer
BCLD	(000452)	Bus Cold Load
TPEF	(000453)	Test Parity Error Freeze Circuits (NonStop II CPU)
SRST	(000455)	Soft Reset (NonStop TXP; NOP on NonStop II)
DDTX	(000456)	DDT Request (NonStop TXP; NOP on NonStop II)
RUS	(000461)	Read micro state (NonStop TXP)
BIKE	(000464)	Bicycle While Idle

Memory Management

MAPS	(000042)	Map in a Segment
UMPS	(000043)	Unmap a Segment (NonStop II processor only)
RMAP	(000066)	Read Map (NonStop II processor only)
SMAP	(000067)	Set Map
CRAX	(000423)	Convert Relative to Absolute Extended Address
RSPT	(000424)	Read Segment Page Table Entry
WSPT	(000425)	Write Segment Page Table Entry
RXBL	(000426)	Read Extended Base and Limit
SXBL	(000427)	Set Extended Base and Limit
LCKX	(000430)	Lock Down Extended Memory
ULKX	(000431)	Unlock Extended Memory
CMRW	(000432)	Correctable Memory Error Read/Write
SVMP	(000441)	Save Map Entries
BNDW	(000450)	Bounds Test Words
SCPV	(000463)	Set Current Process Variables
ASPT	(000470)	Address of Segment Page Table Header

List Management

DLTE	(000054)	Delete Element from List
INSR	(000055)	Insert Element into List
MRL	(000075)	Merge onto Ready List
FTL	(000206)	Find Position in Time List
DTL	(000207)	Determine Time Left for Element

Trace and Memory Breakpoint

TRCE	(000217)	Add Entry to Trace Table
SMBP	(000404)	Set Memory Breakpoint

SECTION 10

GUARDIAN MODULES AND DATA STRUCTURES

This section begins a description of the GUARDIAN operating system. The general approach is to first present an overview of the locations of the various modules and data structures of the operating system, in the context of virtual memory segments, and then to follow this later with descriptions of the functioning of these components.

SEGMENTED ORGANIZATION OF GUARDIAN OPERATING SYSTEM

Figure 10-1 presents an overview of the locations of the major structures of the operating system.

As shown, virtual memory (or "absolute extended memory") consists of 8192 absolute segments numbered from 0 through 8191. These exist primarily as code files and data swap files on the system disc (see lower right corner of the diagram), though various pages of these segments will be present in physical memory at various times. Each segment is 64K words of storage.

The first 128 absolute segments are reserved by the operating system. The first sixteen of these are permanently mapped by the NonStop TXP processor. (The NonStop II processor permanently maps only absolute segments 1, 3, and 6 through 13.) The allocations for all the processes that exist in a given processor at a given time begin at segment 128. Usually, the first several processes will be GUARDIAN operating system processes.

The following paragraphs describe the present allocations of absolute segments.

Segment 0 is used by the XRAY peformance monitor software, and contains the XRAY counters.

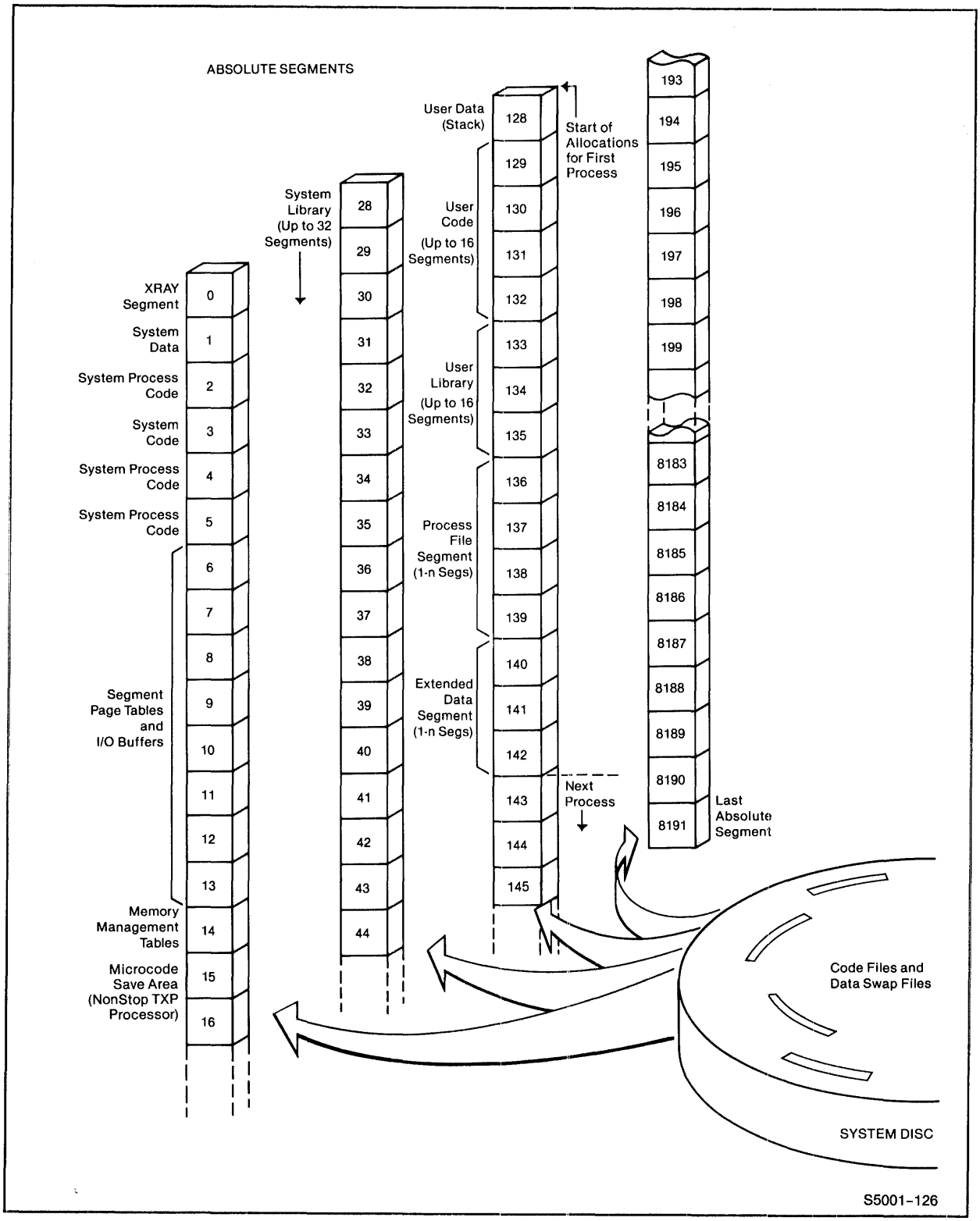


Figure 10-1. Locations of Major Software Structures

Segment 1 is the system data segment, which contains most of the major system tables; accordingly, the contents of this segment are referenced frequently. Segments 2, 4, and 5 are allocated by SYSGEN to system process code segments that are frequently referenced and (on a NonStop TXP processor) will benefit from being permanently mapped.

Segment 3, the system code segment, contains all interrupt handlers and the most frequently used system procedures; less frequently used procedures are relegated to the system library, which is the set of segments beginning at segment 28 (%34).

Segments 6 through 13 are used to store all of the page tables for every segment currently allocated in the processor, and also are used as storage space for I/O buffers. Segment 14 is used to store memory management tables other than the page tables. Segment 15 is used by the NonStop TXP processor to save loadable portions of CPU microcode in the event of power failure.

Segments 16 through 127 (mostly not listed in Figure 10-1) are used for the following purposes:

16	(%20)	Process Control Block Extension, Subchannel Table Extension (PCBX, SCTX)
17	(%21)	Destination Control Table (DCT)
18	(%22)	Page-Process Indentification Table (PAGEPIN)
19	(%23)	Network Routing Table (NRT)
20	(%24)	System library Entry Point Table (SEP)
21	(%25)	Extended System Pool
22	(%26)	DST Transition Table
23 through 27	(%27) through (%33)	Reserved
28 through 92	(%34) through (%133)	System library (SL) segments (see Figure 10-1) (First 32 presently allocatable)
93 through 120	(%134) through (%170)	Reserved
121 through 124	(%171) through (%174)	Debug stack segments
125	(%175)	Messenger process's System Status Message Buffer
126	(%176)	Reserved
127	(%177)	Microcode save area (NonStop II processor)

Beginning at segment 128 (%200) are the segment allocations for the first process. The example in Figure 10-1 shows 15 segments allocated to the first process--with the second process beginning at segment 143. The allocations shown in this example are not necessarily typical, but they do illustrate the allocations that can be made for a process.

The first allocation for a given process is the user data segment (the process stack segment). This is followed by one or more segments of user code--not necessarily in numerical sequence. These segments comprise the main program code of the process. Optionally, there can also be up to 16 segments of user library--that is, a collection of procedures that are privately callable only within this process. (The user code and user library may not need to be allocated for some processes if these can be shared with another process in this CPU.)

As the process executes, there may be additional allocations for extended data segments. Any such segment (it is viewed as one logical segment by the user) may be made up of any practical number of contiguous absolute segments. (The block of segments need not be contiguous with the other segment allocations of the process--though shown this way for simplicity.) For each process, one extended data segment is always present, automatically assigned by the operating system: the process file segment. Other extended data segments are optional, being present only when specifically requested by the process. The process file segment is used by the file system to track the status of communication with every file that is opened by the process.

It should be noted that there is not necessarily a one-to-one correspondence between segments and disc files. A single code file for a process's user code may be up to 16 absolute segments in length, and the file for an extended data segment may be even longer. Conversely, the user code segment allocation for many similar processes (for instance, command interpreters) may all correspond to the same shared code file on the system disc.

Having viewed the overall storage arrangement for the GUARDIAN operating system, we can now observe the methods used to access these system structures. As noted previously, nonprivileged processes cannot directly use absolute addresses. Consequently, absolute segment numbers are meaningless in the user process context. Instead, these processes use relative addressing, which allows access to only a limited set of segments. This both simplifies the addressing requirements and protects each process from all others.

Figure 10-2 illustrates one example of process access to a frequently used absolute segment, system data. System data, as observed in the preceding figure, is absolute segment 1. This segment is accessed by the system (on behalf of any process) either as Short Address Space 1 (using 16-bit addresses) or as relative segment 1 (using 32-bit addresses). (This happens to be the only example of direct numerical correspondence of address spaces.) Other examples are illustrated in the succeeding figures.

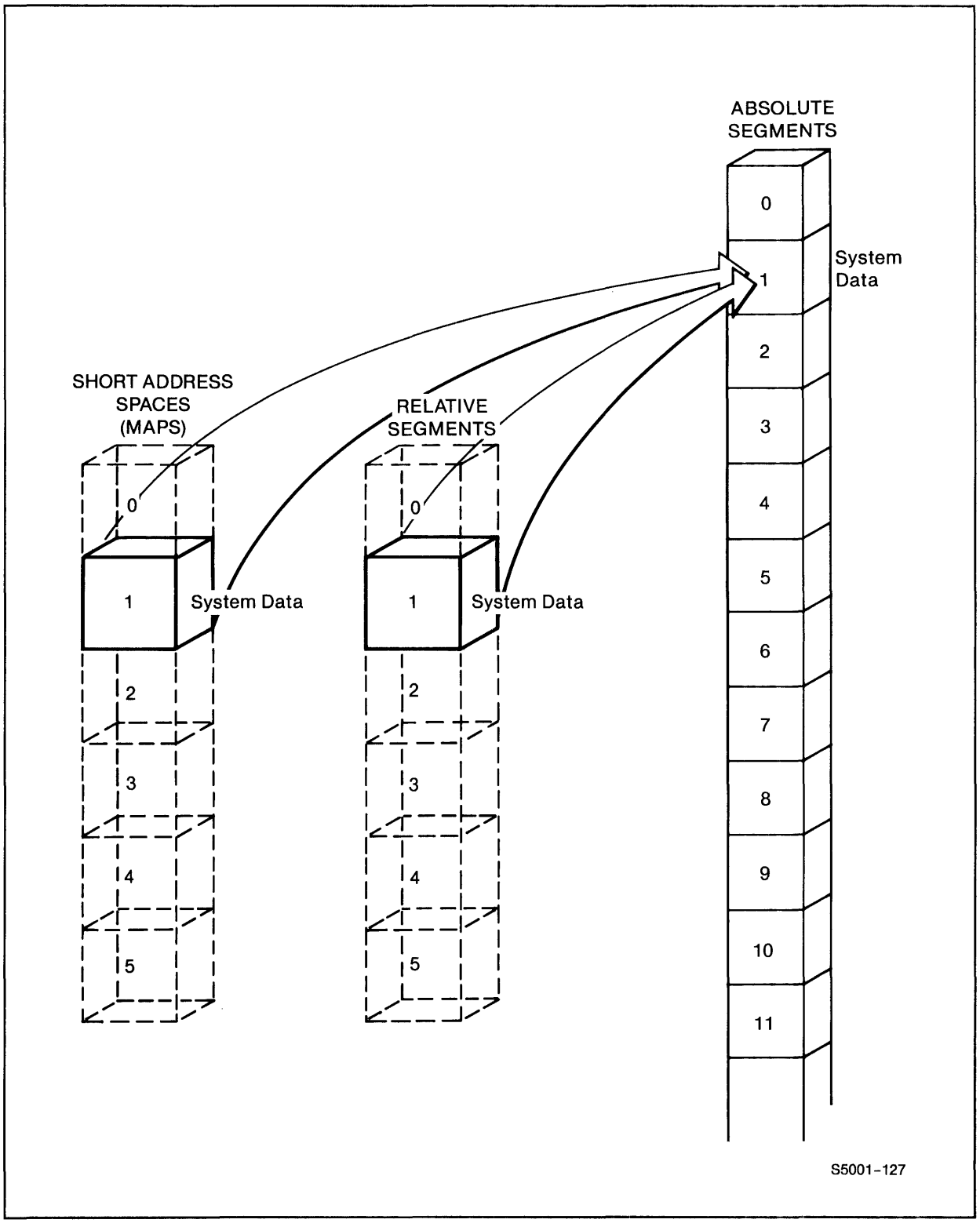


Figure 10-2. Access to System Data Structures

Figure 10-3 illustrates user access to the system procedures in the system code and system library segments. The single system code segment, absolute segment 3, is accessed by the system (on behalf of any process) as Short Address Space 3. The space ID for this segment is SC.0.

The 32 system library segments, absolute segments 28 through 59, are accessed by the system (on behalf of any process) as Short Address Space 5. The specific segment selected depends on the current space ID value, which can range (octally) from SL.0 for segment 28 up to SL.%37 for segment 59.

The system code segment and all system library segments correspond to separate portions of the OSIMAGE file on the system disc. This is depicted on the right side of Figure 10-3.

System code and system library segments may also be accessed with relative extended addresses, but only as part of the "current code" arrangement that will be described later in connection with Figure 10-5.

Figure 10-4 illustrates short-address access for a process to its own code, data, and library segments. (Relative extended-address access will be considered in the subsequent figure.)

User data (that is, the process stack segment) is accessed by the system (on behalf of the process) as Short Address Space 0. This corresponds to some allocated absolute segment, illustrated as segment 422 in the figure. Physically, this segment consists of a data swap file on the system disc.

Current user code is one of the user code segments, illustrated as segments 423 through 426 in the figure. The selected segment is individually identified by a space ID value in the range of UC.0 through UC.%17; UC.2 is assumed in the figure. This segment is accessed by the system (on behalf of the process) as Short Address Space 2. Physically, it consists of a portion of the main program code file (object file) for the process.

The User Library is illustrated as segments 427 through 430 in the figure. One of these is "current" if the process is presently executing a user library procedure. The selected segment is individually identified by a space ID value in the range of UL.0 through UL.%17. UL.1 is assumed in the figure. This segment is accessed by the system (on behalf of the process) as Short Address Space 4. Physically, it consists of a portion of the user library file (object file) which is bound to the main program code file for the process. A "demand mapping" scheme is used for such segments--that is, a user library segment is mapped only by a procedure call to (or exit from) that segment.

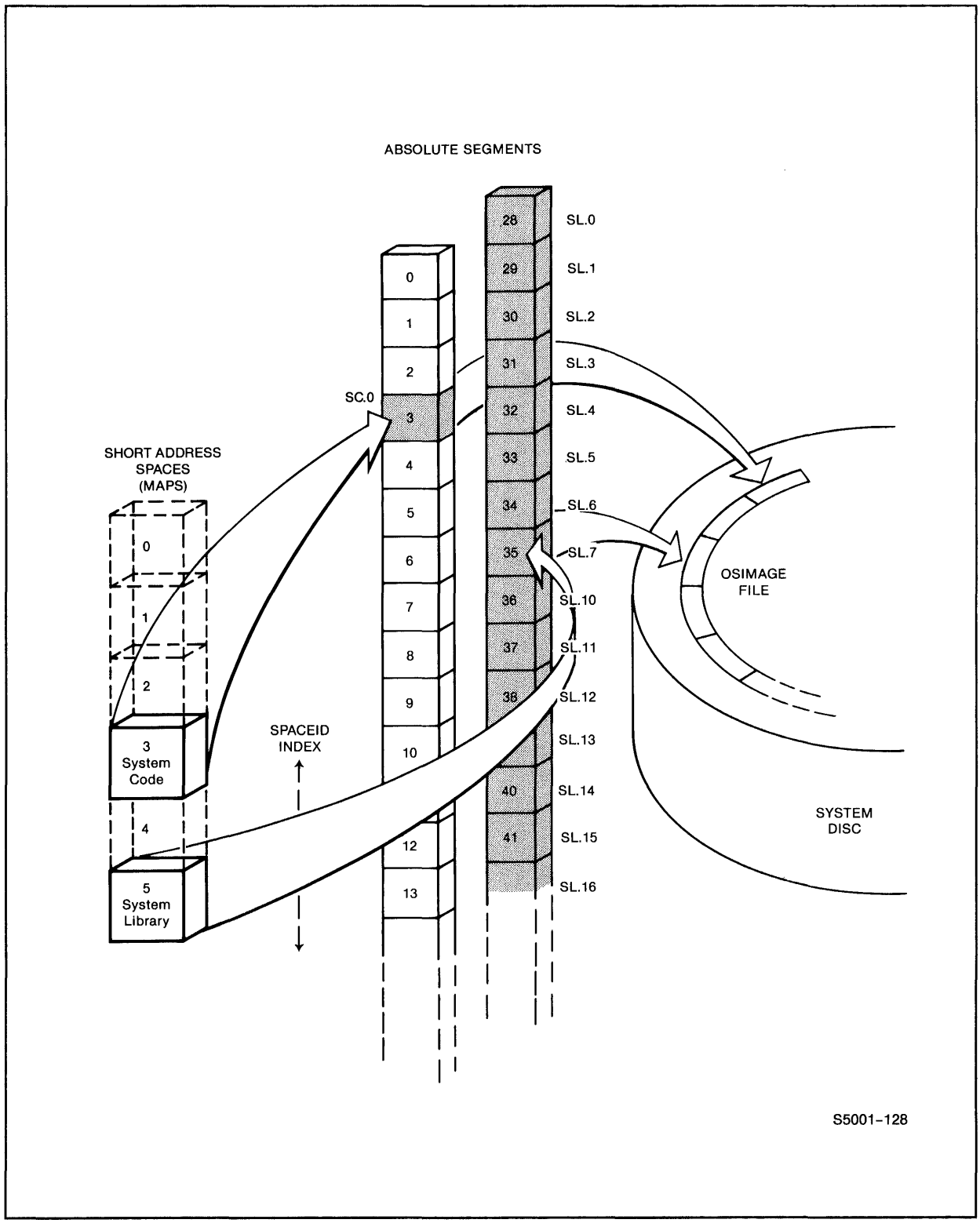


Figure 10-3. Access to System Procedures

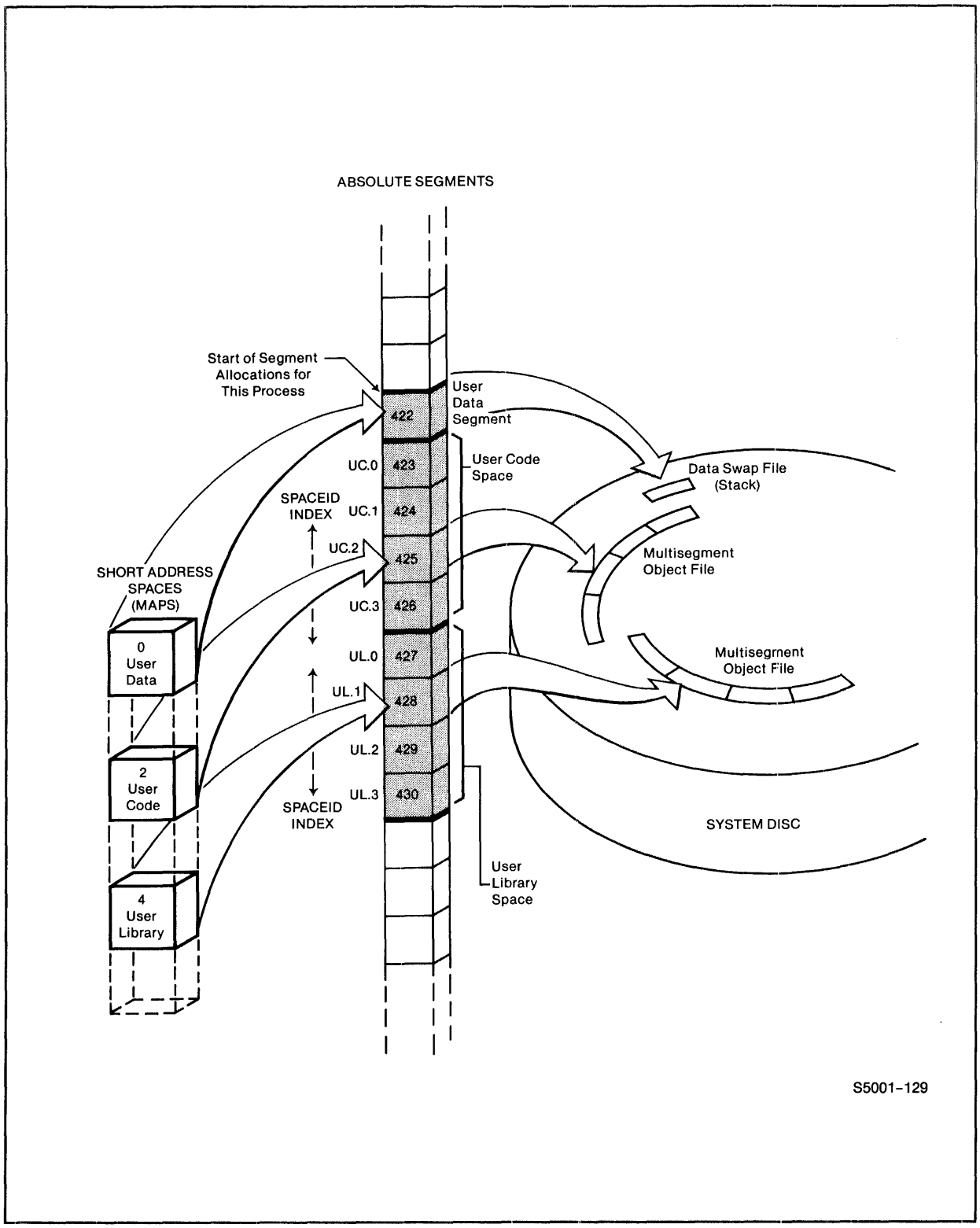


Figure 10-4. Short-Address Access to Process Code and Data

Figure 10-5 illustrates the case of relative extended address access to the code segments that are accessible to a given process. In this case, "current code" and "user code" can be accessed as relative segments 2 and 3 respectively. The specific absolute segments selected for these functions at any given time are dependent on the current space ID value.

For current code (relative segment 2), there are two levels of selection ("switching"). One level selects the space (SC, SL, UL, UC); this is done in the Environment register (ENV). The second level is the space ID index value (0 - %17 for UL and UC, 0 - %37 for SL, and 0 always for SC). For illustrative convenience, the absolute segment numbers shown in Figure 10-5 correspond to the numbers used in the preceding two figures. The correspondence to disc files is not shown; this was illustrated in those preceding figures.

For user code (relative segment 3), the space is predetermined: UC. The space ID index value selects one of the segments allocated to the current process.

For performance reasons, no indexing occurs for system code. The space ID is always SC.0, and the corresponding absolute segment (3) is permanently mapped.

Figure 10-6 illustrates the case of access to the Process File Segment. This is an extended data segment belonging to the process; like any other extended data segment it can be addressed only by relative extended addresses--beginning at relative segment 4. The process file segment, however, is different from other extended data segments in that it cannot be accessed by nonprivileged users; access is permitted only for privileged or callable procedures. For access to be possible, the process file segment must be the "current" extended data segment.

The foregoing five illustrations have outlined the general layout and access for most of the GUARDIAN operating system components originally presented in Figure 10-1. The accessing of some structures, such as I/O buffers, Segment Page Tables, and other memory management tables has not been shown, since these are addressable only by absolute extended addresses referring to the absolute segments.

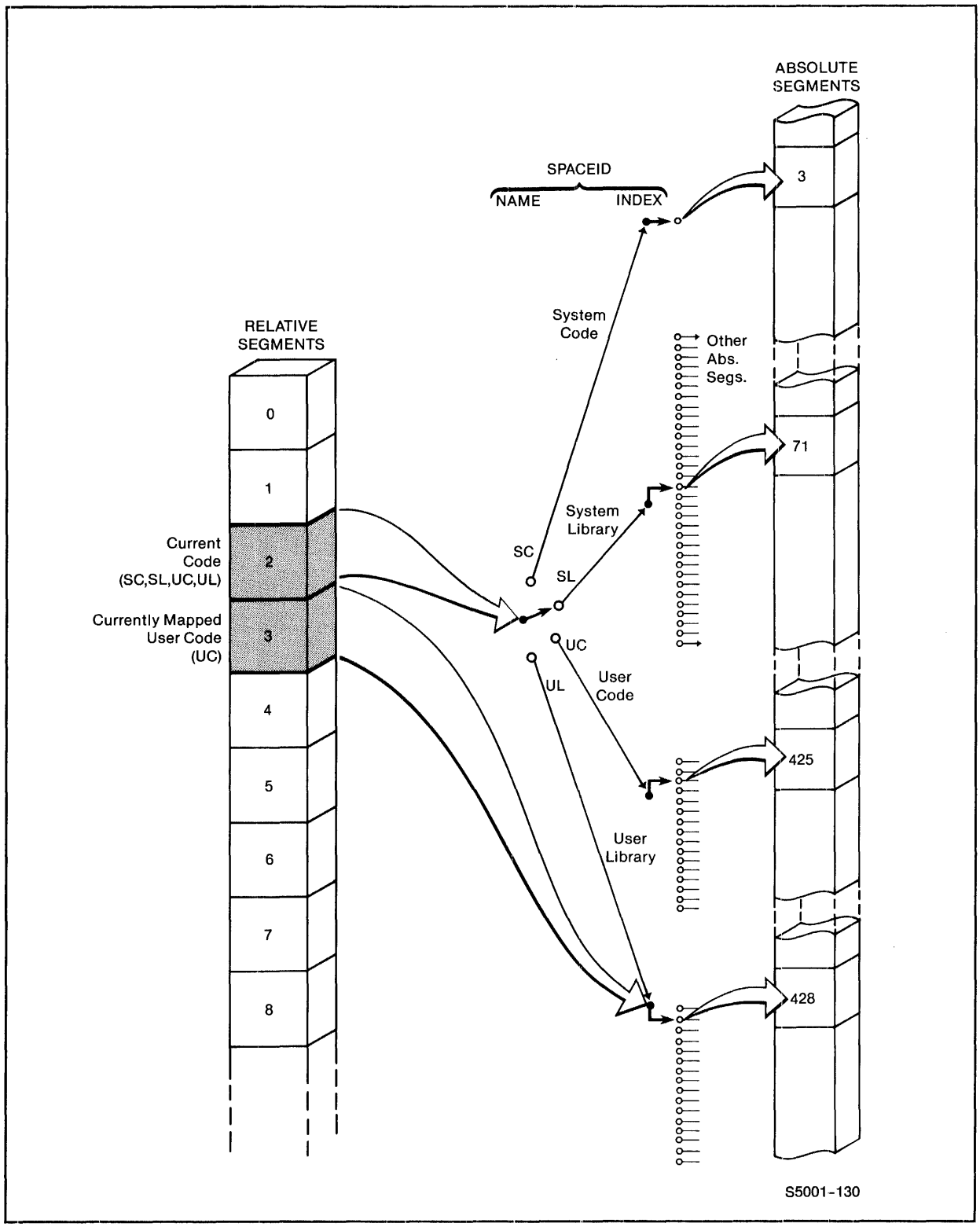


Figure 10-5. Extended-Address Access to Code Segments

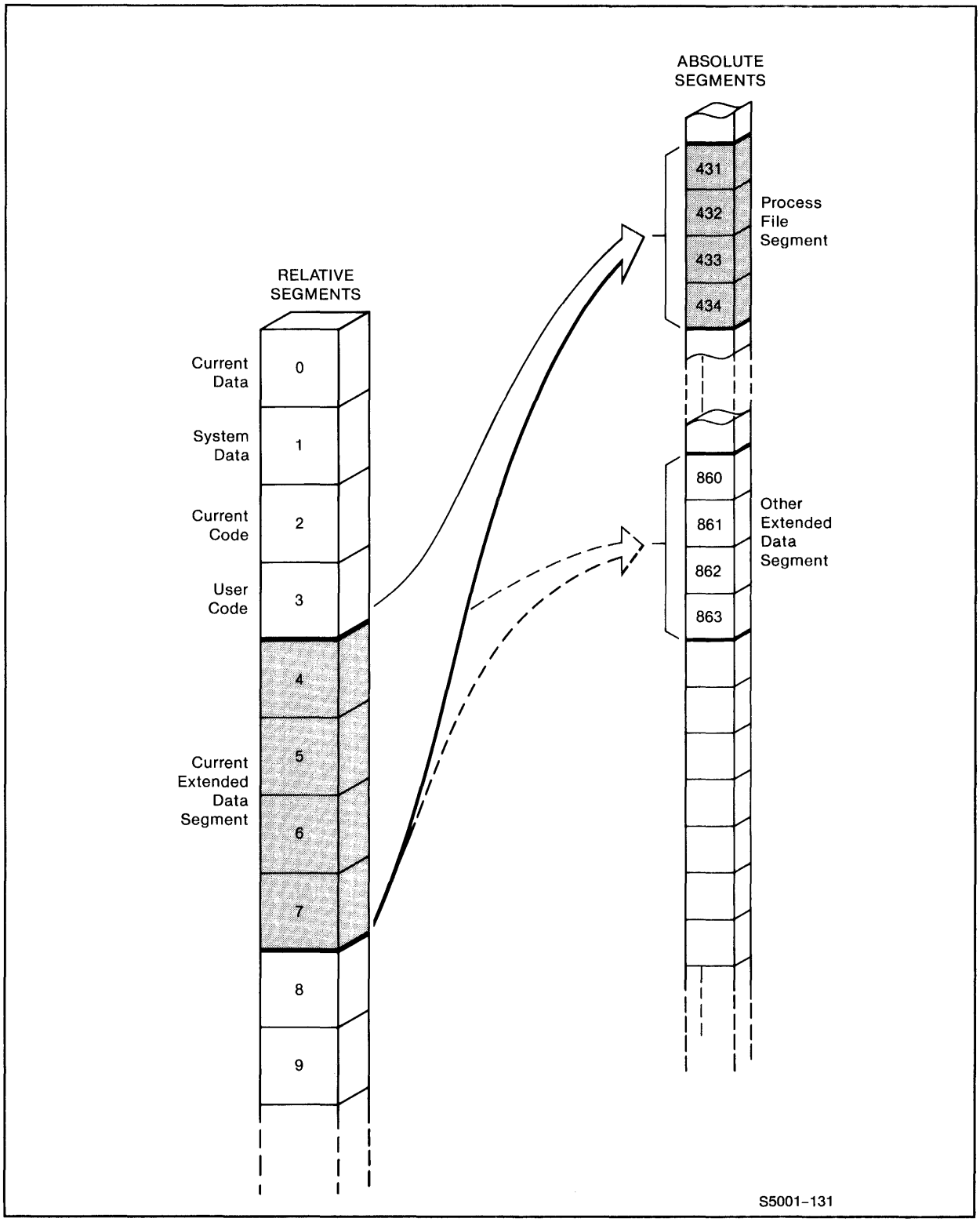


Figure 10-6. Access to Process File Segment

SECTION 11

THE PROCESS ENVIRONMENT

At any instant of time, the processor is operating in one of two fundamental environments: either the process environment or the interrupt environment. This section describes the process environment.

PROCESS DEFINITION

In the Tandem NonStop system, a process is created either by entering the RUN command or by programmatically calling the NEWPROCESS procedure. As illustrated in Figure 11-1, each process consists of the following:

- An unmodifiable code area that contains instructions
- A separate, private data area called a stack
- A system table entry called the Process Control Block (PCB) that defines the state of the process in the system. (The Process Control Block Extension, PCBX, which contains less frequently used information pertaining to the process, is considered to be a logical part of the PCB.)

Each time a user requests program execution, a process is created. Thus, if a user runs two separate programs, the GUARDIAN operating system creates two corresponding processes. And if he runs the same program twice, or two users run the same program concurrently, again two processes are created. Neither process "owns" the program code; code is sharable among processes within the same processor. The data, however, is private to the process.

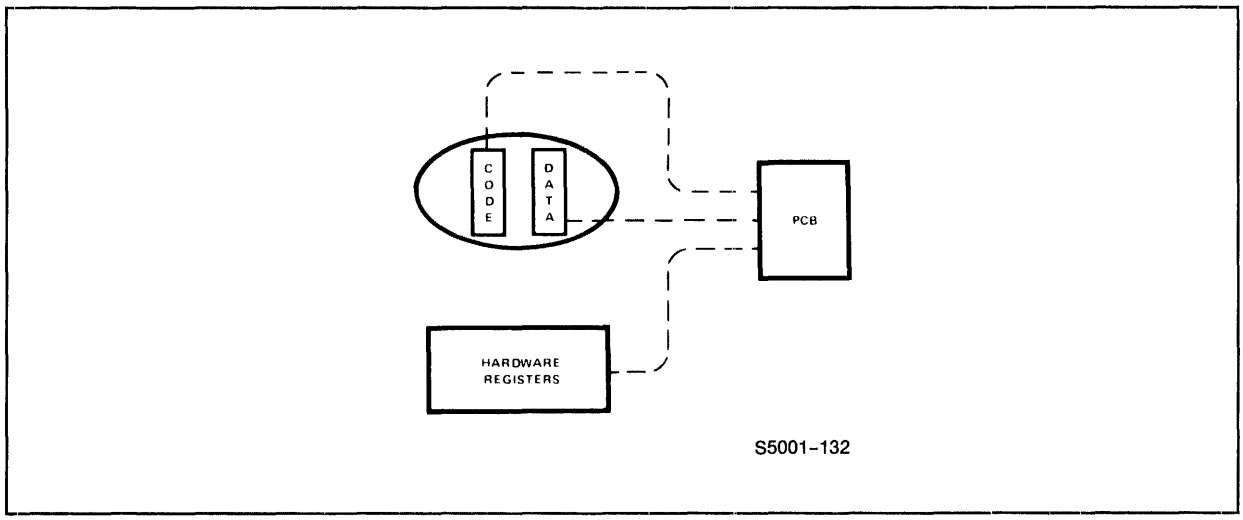


Figure 11-1. Elements of a Process

Up to 256 processes can be executing concurrently in each CPU. Although these processes share some resources, such as main memory, only one process can be executing at any instant.

A process executes until either: 1) it must wait for a resource or for a message or an I/O operation; or 2) a higher priority process becomes ready to execute. The GUARDIAN operating system then saves the process state (the space ID of the currently executing code segment, and the contents of registers P, ENV, L, S, and R0:R7) in the process's PCB, and then chooses a new process to execute, if one is ready. That process's state is taken from its PCB and used to continue its execution from the point where it was last executing.

Application processes (that is, all processes that are not system processes) have only a temporary existence--they are subject to a "life cycle" that has as its phases: creation, execution, and termination (see Figure 11-2).

System processes, on the other hand, have all the same physical characteristics of application processes, but are a permanent part of the system. They are automatically executed when a CPU is loaded.

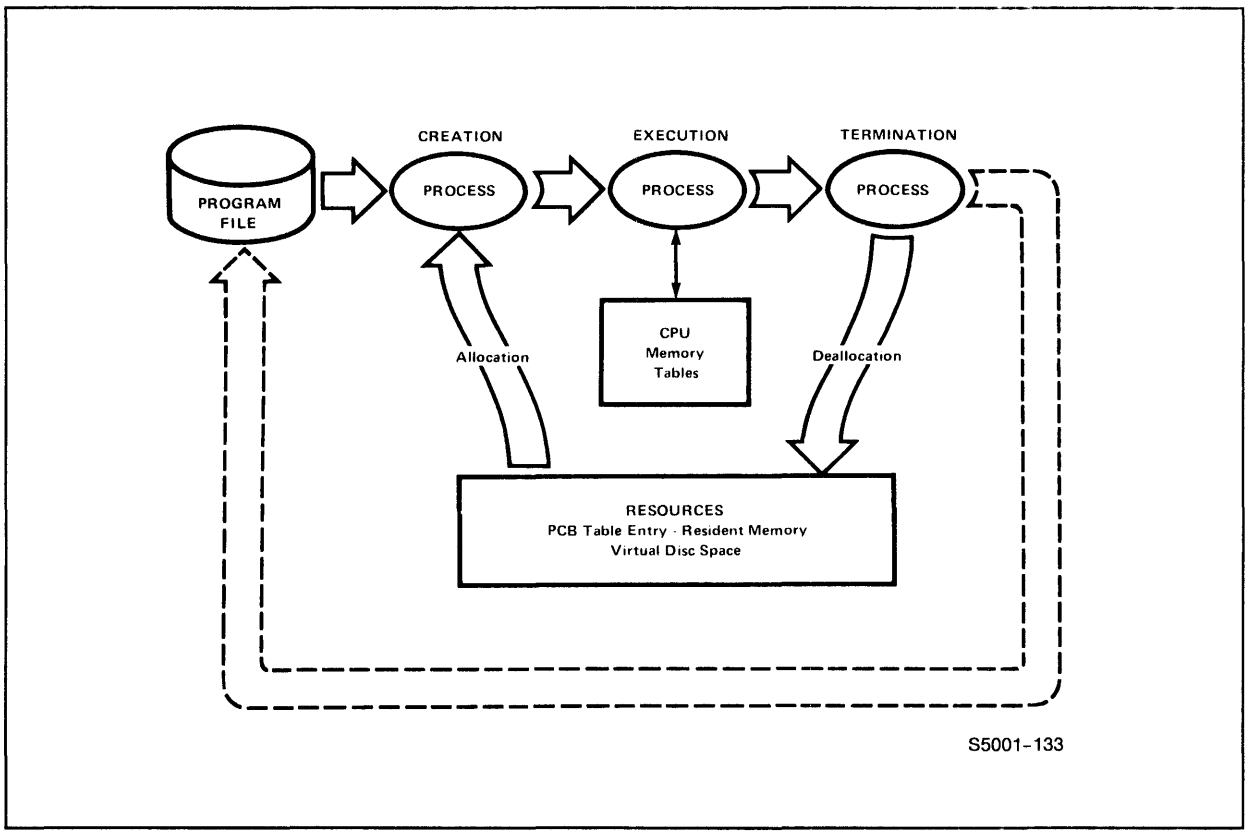


Figure 11-2. Process Creation, Execution, and Termination

SYSTEM PROCESS CREATION

Because of their permanent nature, system processes must be created in a slightly different way than application processes. The following discussion and Figures 11-3 and 11-4 show how this is done.

The system processes for a new GUARDIAN operating system configuration are created by running the SYSGEN program (Figure 11-3). SYSGEN executes as a user process and is run in the same way as any other process. SYSGEN reads information about the new configuration and builds a complete image of the new system, storing it in a file named OSIMAGE.

After it has built the image of the system, SYSGEN writes a system image tape (SIT), containing the OSIMAGE and other files. This tape can be used in two different ways:

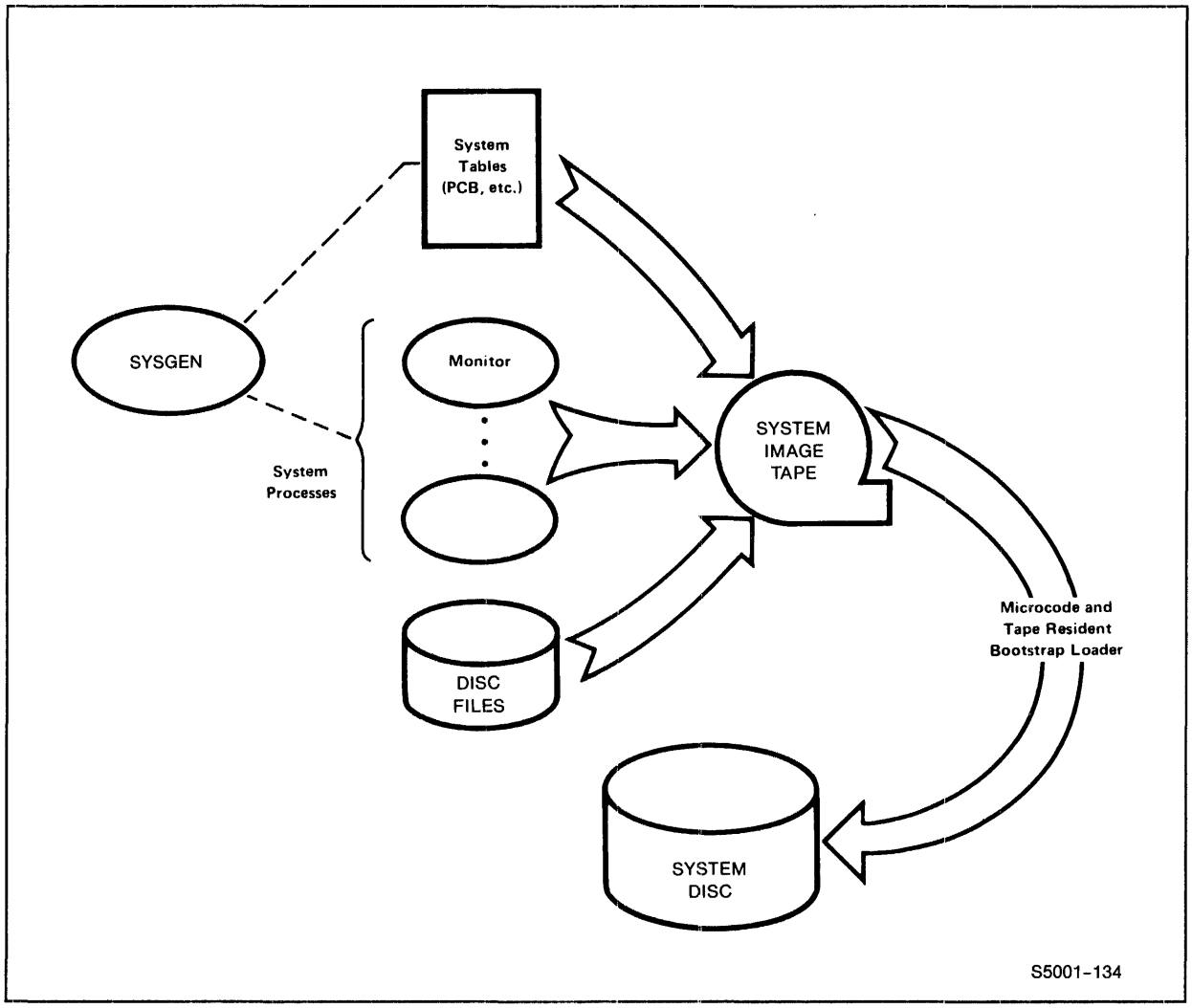


Figure 11-3. System Configuration and Loading--Part 1

1. On a new system, the tape can be "loaded" onto the system disc. The tape contains a bootstrap program which simply reads the rest of the tape and writes it to disc. This destroys any previous information on the disc, so this is usually done only once, when the system is delivered.

2. On a running system, the SIT files can be "restored" to disc, using the RESTORE utility program. (Except for tape and disc bootstraps, the SIT has the same format as a backup tape.)

The OSIMAGE and many other files reside in some SYSnn subvolume (where nn is a 2-digit octal number). Once these files are on disc, the operator can cold-load the system by setting the nn value and a disc unit address in the processor switches. This may be done using the physical switches, but it is more convenient to use the OSP (Operations and Service Processor) if available.

When the operator turns the key switch to LOAD (or performs the equivalent operation at the OSP), the CPU finds the image, copies it to memory, and then begins execution in the interrupt environment. Once the kernel has been initialized, the Dispatcher begins executing the system processes which were placed in the image by SYSGEN.

The monitor process opens the OSIMAGE file, since this file also contains code and data which is nonresident and must be fetched by the memory manager when needed. The monitor starts a command interpreter process, which can be used by the operator to continue loading the system.

In the normal case, the operator sets the time, reloads the remaining CPUs and starts additional command interpreters and application programs.

Only one CPU is loaded from disc (cold-loaded). The remaining CPUs are loaded using the RELOAD program, which reads OSIMAGE and transfers the image across the interprocessor buses. This "bus load" can load all down CPUs in parallel.

In Figure 11-4, the initial command interpreter reads input from a command (OBEY) file that creates all other command interpreter processes that run on the system. A production shop, however, might never start a command interpreter in another terminal, but instead might use the initial command interpreter to simply run application processes. (For reliable operation, however, it is recommended to have at least two command interpreters running; otherwise a simple terminal failure could necessitate a cold load.)

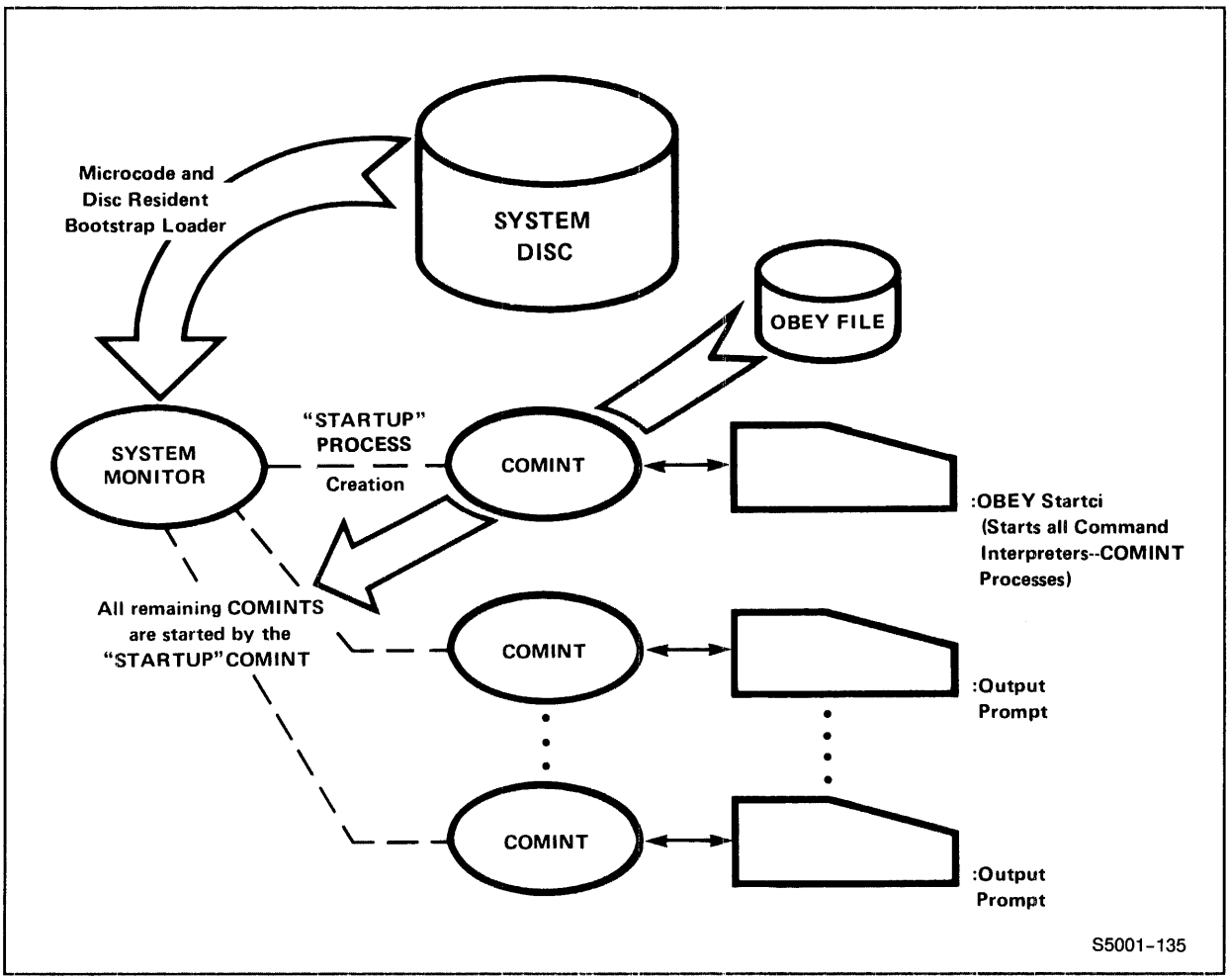


Figure 11-4. System Configuration and Loading--Part 2

APPLICATION PROCESS CREATION

Once a command interpreter is started on a terminal, users may log onto the system to create and run their application processes. The following example shows how a user might construct an application process. This example is divided into twelve steps, a through l; it illustrates source code entry, compilation, and object file execution.

a. When the command interpreter is ready to accept input from a user, it displays a colon (:) as a prompt. When the user first sits down at his terminal, he must respond to this prompt with a LOGON command to gain access to the system (Figure 11-5). The command interpreter then displays:

   - Current version number and date of the GUARDIAN operating system
   - Its primary CPU and (if any) backup CPU
   - Present date and time
   - Prompt for the next command.

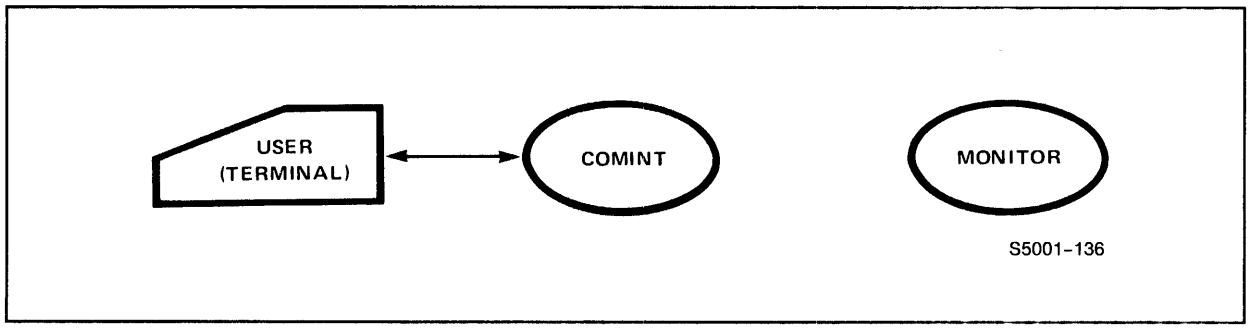


Figure 11-5. Logging On to GUARDIAN Operating System

b. Now the user enters an EDIT command to obtain the services of the text editor process. Since this process does not yet exist, the command interpreter sends a process-creation request to the monitor process. (From this point on, the command interpreter typically waits until the editor process terminates before resuming execution.) The monitor uses the code in the EDIT program file to create the editor process (Figure 11-6). Because the command interpreter is waiting, the user's terminal now appears to belong exclusively to the editor process.

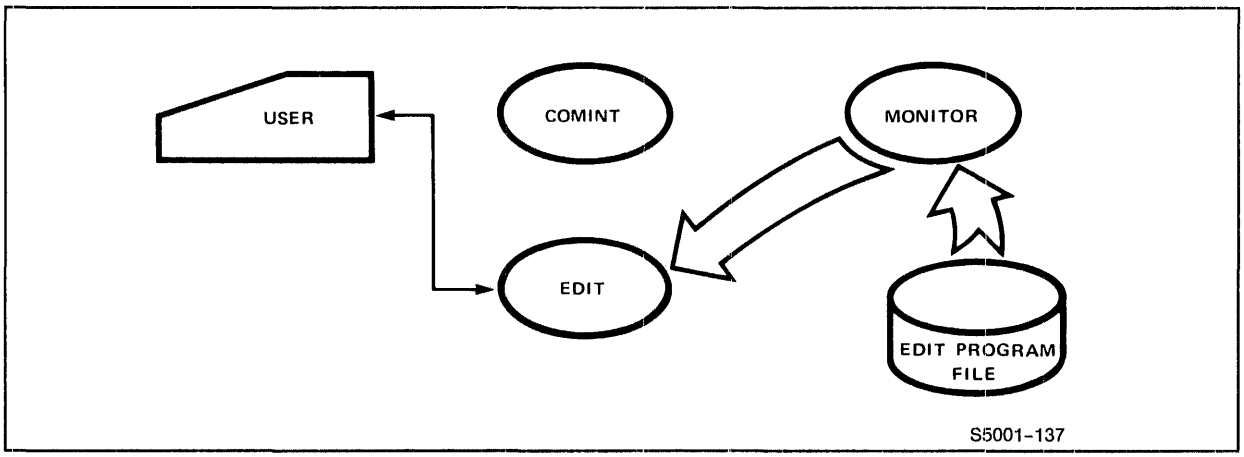


Figure 11-6. Creating the Editor Process

c. When the editor begins execution, it displays its name and current version number on the user's terminal, followed by an asterisk (*) as a prompt for an editor command. The user then enters commands to create a new, empty disc file and to place source statements in this file (Figure 11-7). Assume a program is being entered ("edited") in the TAL language.

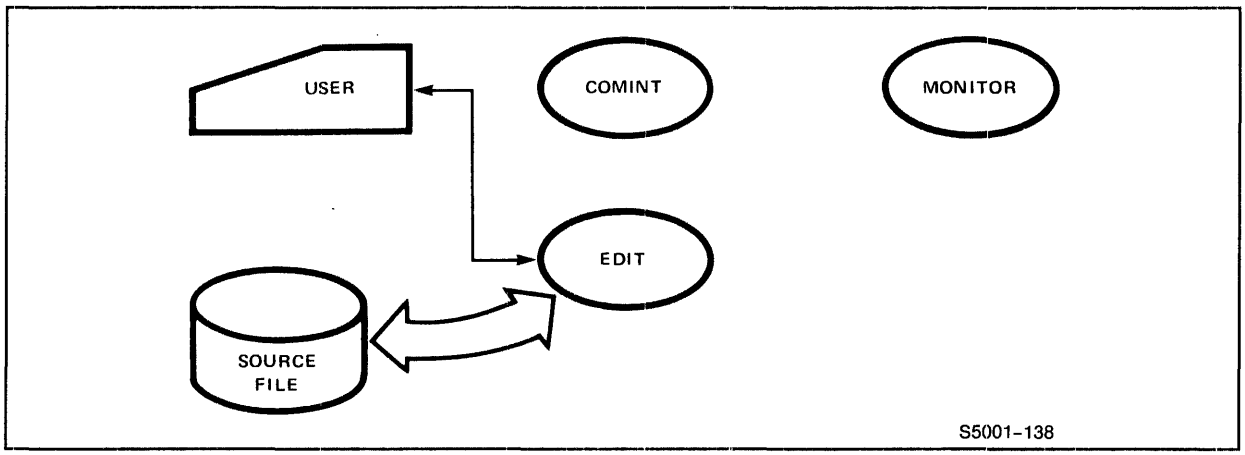


Figure 11-7. Producing an EDIT Text File

d. Next, the user enters an EXIT command to terminate the editor. In response, the editor process sends a termination request to the monitor process. The Monitor terminates the editor and returns that process's resources to the system.

The source file created by the editor, however, remains on disc as a permanent file (Figure 11-8)--its disc space is not returned by the operating system when the Edit process terminates.

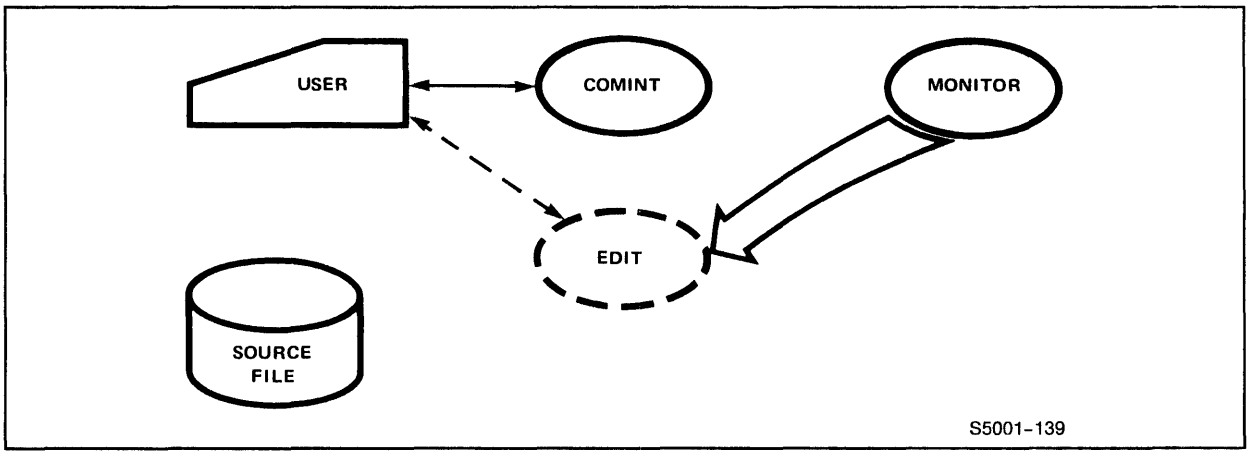


Figure 11-8. Terminating the Editor Process

e. Now the command interpreter resumes control, issuing a prompt for another command. The user responds by entering the TAL command to request the services of the TAL compiler to translate his source file into object code. Again the command interpreter sends a message to the monitor, this time requesting creation of the TAL process (Figure 11-9).

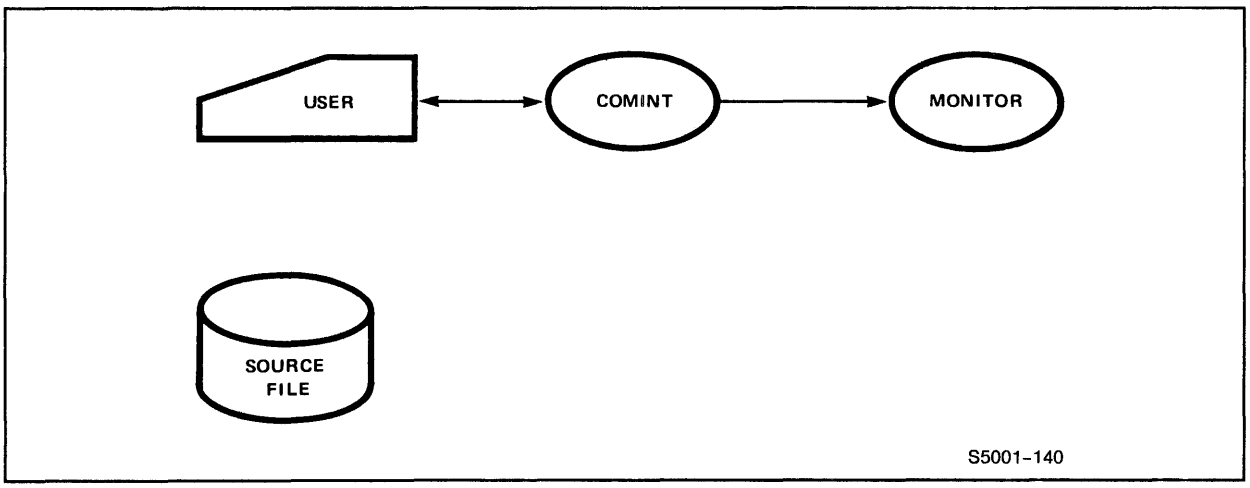


Figure 11-9. Requesting Access to the TAL Compiler

f. Typically, the command interpreter waits during the initialization and execution of the newly created process. The monitor uses the code in the TAL program file to create the TAL process. Like the Edit process, TAL also appears to have sole access to the command interpreter's terminal, which is the default listing device for the compilation (Figure 11-10).

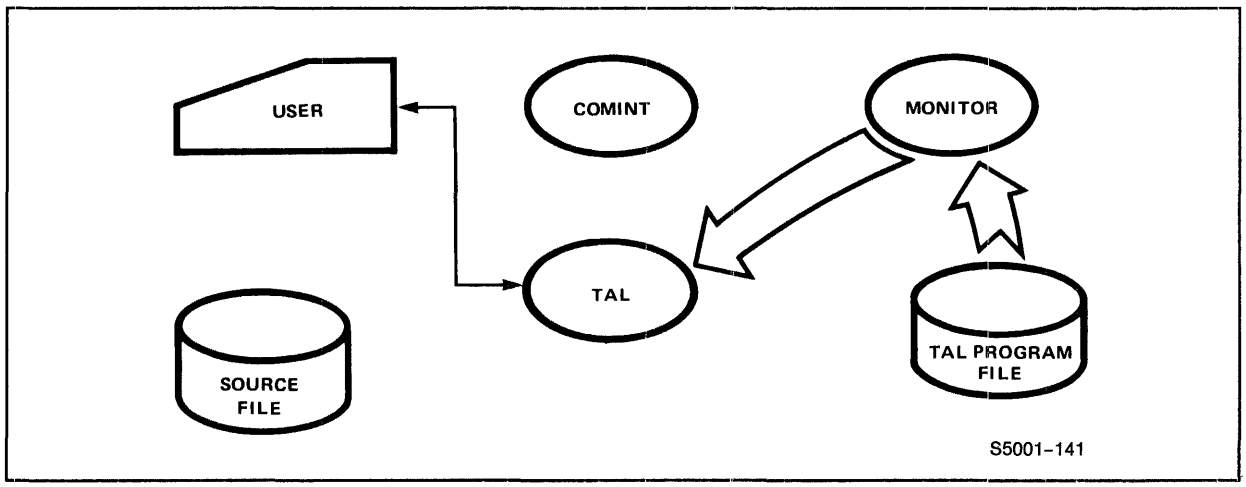


Figure 11-10. Creating the TAL Compiler Process

g. When the user entered the (RUN) TAL command in Step e, the command interpreter transmitted startup information to the TAL compiler. This information directed the compiler to read source images from the EDIT file produced in Step c, and place the compiled code into a specific program file. (Normally, the user supplies the names of both the source and program files as TAL command parameters. Furthermore, the user may specify a particular file to receive the compilation listing--or may, as in this case, omit this specification and receive the listing on his terminal.) Now the compilation takes place, with the resulting object code written to a program file on disc (Figure 11-11).

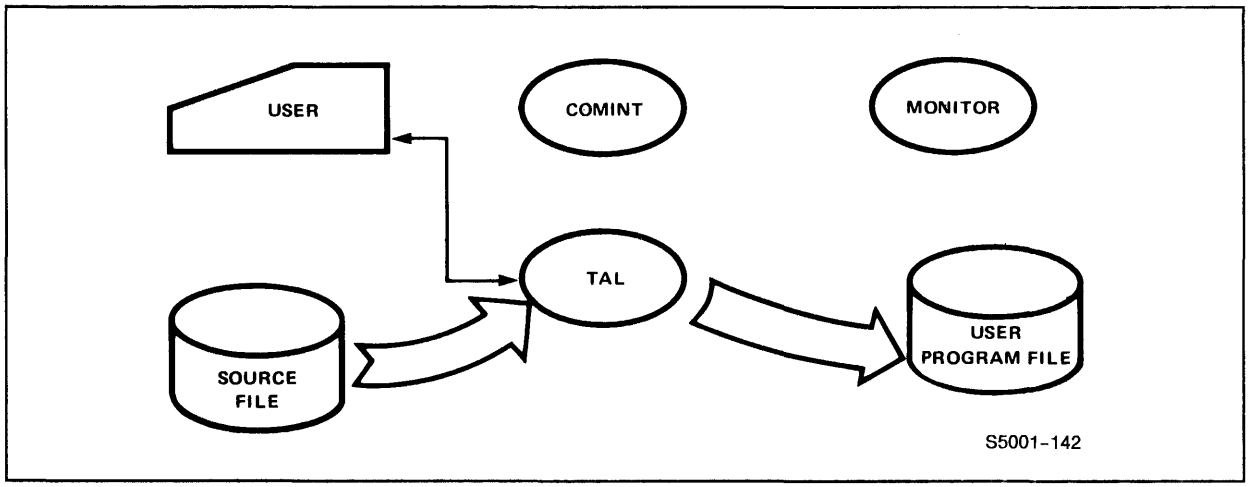


Figure 11-11. Compiling the Source Program into Object Code

h. The termination of the TAL process causes a request to be sent to the monitor, which responds by returning the system resources held by TAL. If the compilation was successful, the program file for the user's application process now exists on disc (Figure 11-12).

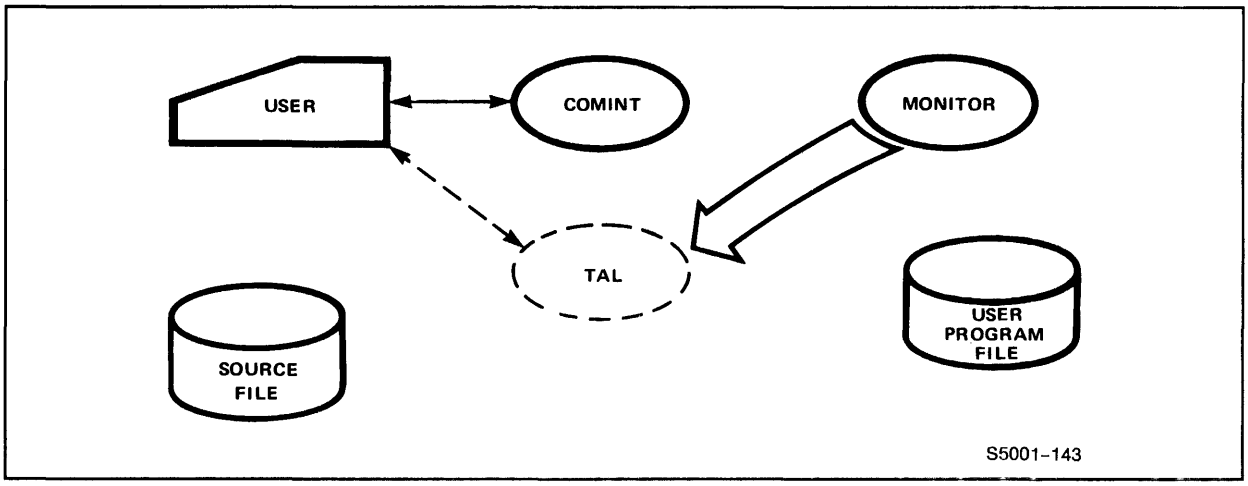


Figure 11-12. Terminating the TAL Process

i. When the TAL process terminates, the command interpreter prompts the user for a new command. (If the compilation was not successful, of course, the user must reedit and recompile his program.) At this point, the user enters a RUN command to execute the application program residing in his program file (Figure 11-13).

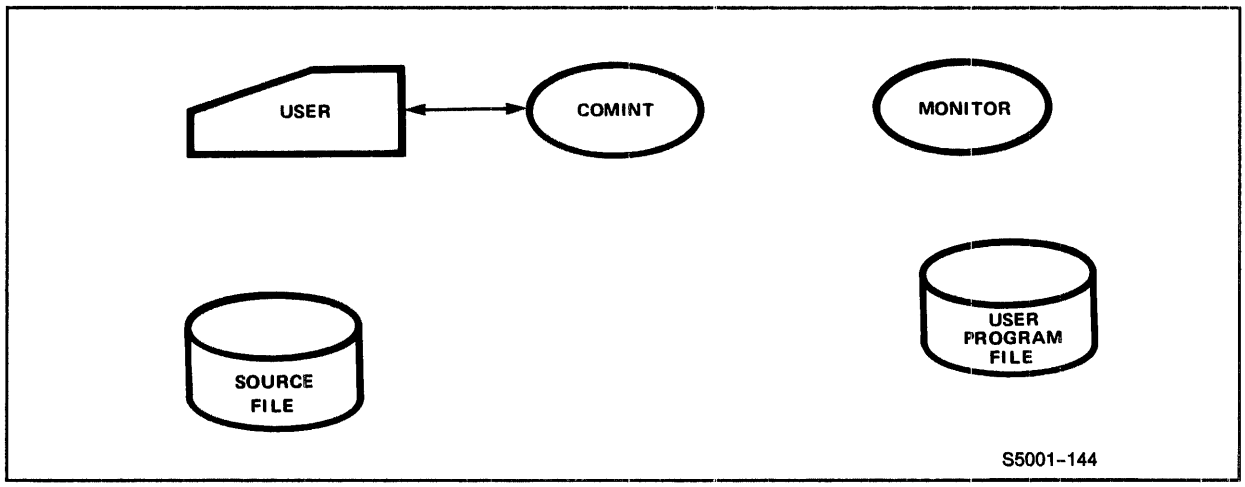


Figure 11-13. Requesting Application Program Execution

j. To run this program, the system creates a new application process in exactly the same way it created the EDIT and TAL processes--in other words, the command interpreter sends a request to the monitor to create a process to run the program (Figure 11-14).

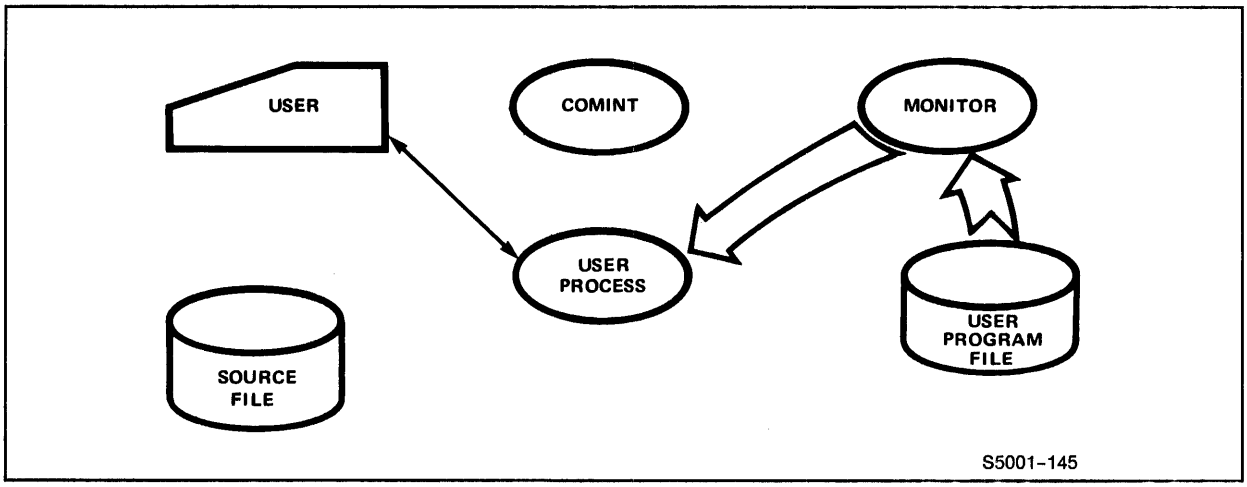


Figure 11-14. Creating the Application Process

Because the user did not specify otherwise in the RUN command, the command interpreter will wait until the application process terminates. (A special RUN command option, however, permits the command interpreter to continue prompting the user while the requested process is running; this option is discussed later.) Because the command interpreter is waiting, the user's terminal is available for use by the application process. As far as the monitor is concerned, the only difference between EDIT, TAL, and the application process is that each of these processes is associated with a different program file. Essentially, they are ALL user processes that the monitor treats in the same way.

k. When the user's application terminates, a request is sent to the monitor, which returns all system resources held by that process (Figure 11-15).

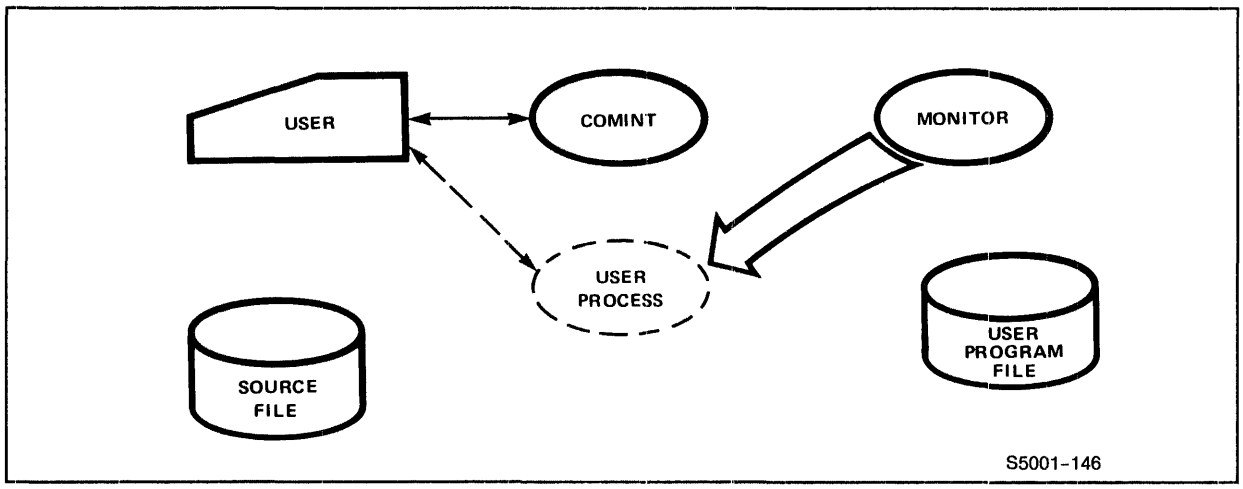


Figure 11-15. Terminating the Application Process

1. When the application process terminates, the command interpreter prompts the user for another command (Figure 11-16).

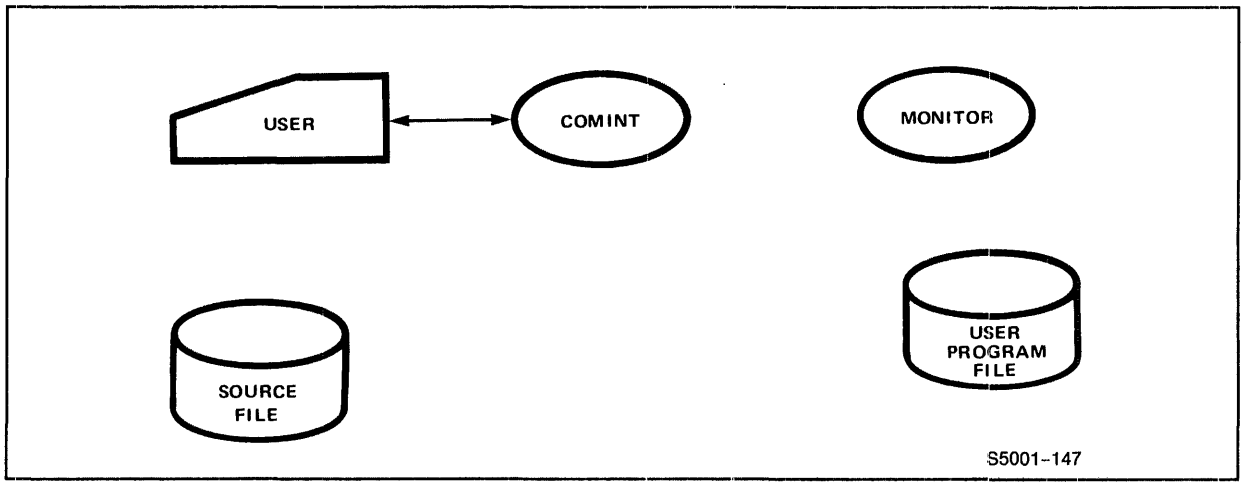


Figure 11-16. Returning Control to the Command Interpreter

Now, the situation is exactly the same as it was before the user created the Edit process--except for the presence of the newly-created source and object files in the system. At this point, the user may request any other system operation he wishes (including logging off).

MULTIPLE APPLICATION PROCESSES

In the previous example, before it prompted for another command, the command interpreter always waited for the termination of the process which it created. It is also possible, however, to create multiple processes without causing the command interpreter to wait for the new process's completion. To do this, a user enters the nowait parameter in the RUN command (Figure 11-17). The command interpreter then creates the requested process and prompts the user for another command. In actuality, the prompt is not issued until certain messages have been passed from the command interpreter to the newly created process. One of these messages is the Newprocess message; another is the Startup message, which contains the names of the input and output files that the new process may open and use. If the user does not specify such file names in the IN and OUT parameters of the RUN command, the command interpreter passes its default file names to the process. (Usually, these default values specify the command interpreter's home terminal.) The ability to tell a new process which files to use for input and output provides great flexibility and makes multiple process creation by the same command interpreter truly useful. Now all processes need not share the same terminal--instead, they can each be assigned a different one.

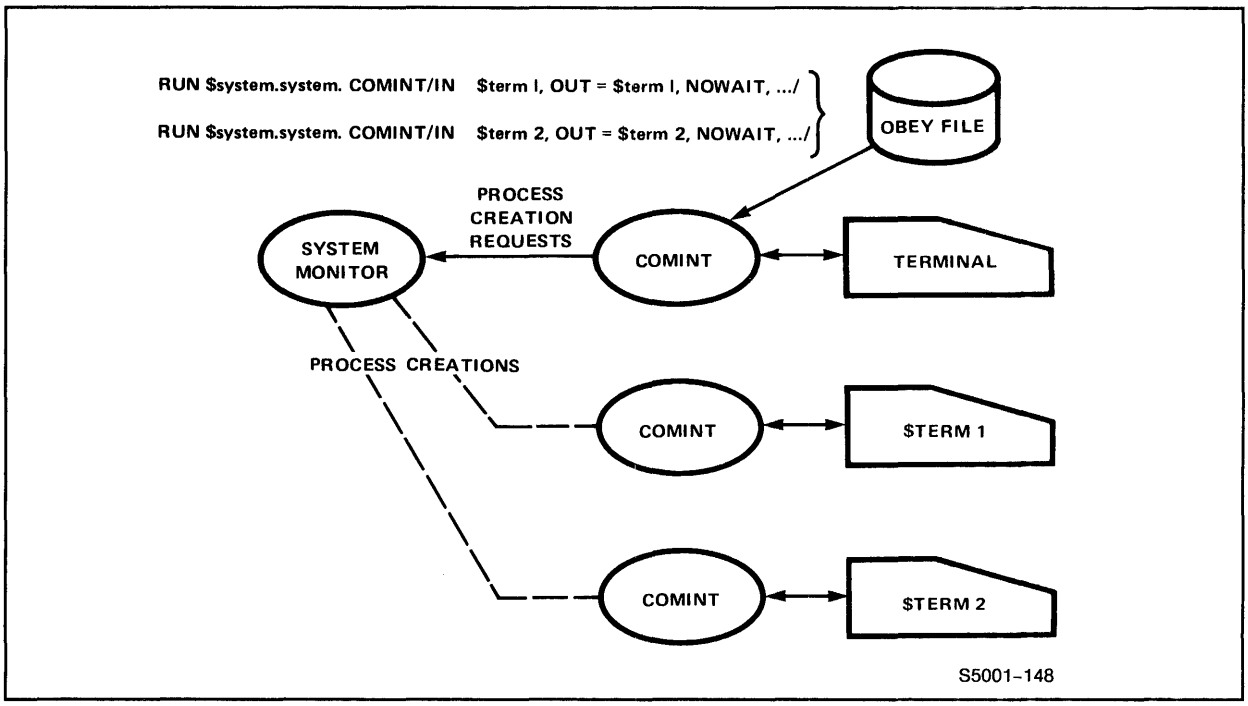


Figure 11-17. Command Interpreter File Assignments

A practical example of one process creating numerous others is the startup of a series of command interpreters. (See Figure 11-17 again.) These are created when the original command interpreter reads and executes a series of commands in an OBEY file. Each RUN command in this file starts another command interpreter and assigns it a unique terminal to use for input-output. Of course, to create concurrent processes, each RUN command must include the nowait parameter. The command interpreters created in this way are treated by the GUARDIAN operating system as ordinary user processes. In fact, the operating system cannot distinguish them from such processes. Any operation done to create these command interpreters could as well be done in the creation of a user application process. Not only can users create application processes with the nowait option and run these processes concurrently--they can also specify which input-output files they should open and use. These files need not be limited to terminals, but may include disc files, other I/O devices, or even other processes (depending on restrictions imposed by the applications).

PROCESS LIFE CYCLE

As mentioned earlier (see Figure 11-2) a process's "life cycle" begins when the process is created from a program file and ends when the process terminates.

The following example, illustrated in Figure 11-18, describes the life cycle of a simple (not necessarily typical) process. Assume that this process is created in a waited manner--that is, once the command interpreter creates the user process, it waits until the user process terminates.

The individual life-cycle events are described in Steps a through h below, which are keyed to Figure 11-18. The sequence of life-cycle events begins at the top of this figure and proceeds to the bottom. Each column in this figure represents one of the five fundamental system processes involved in the user process life cycle, and summarizes the operations performed when that process executes. (Blank areas in a column indicate periods when the process is not running.) The broken-line boxes in the disc process column indicate that it may be executing in another CPU. And finally, each arrow in the figure represents a message used to communicate with, or pass requests between, cooperating processes.

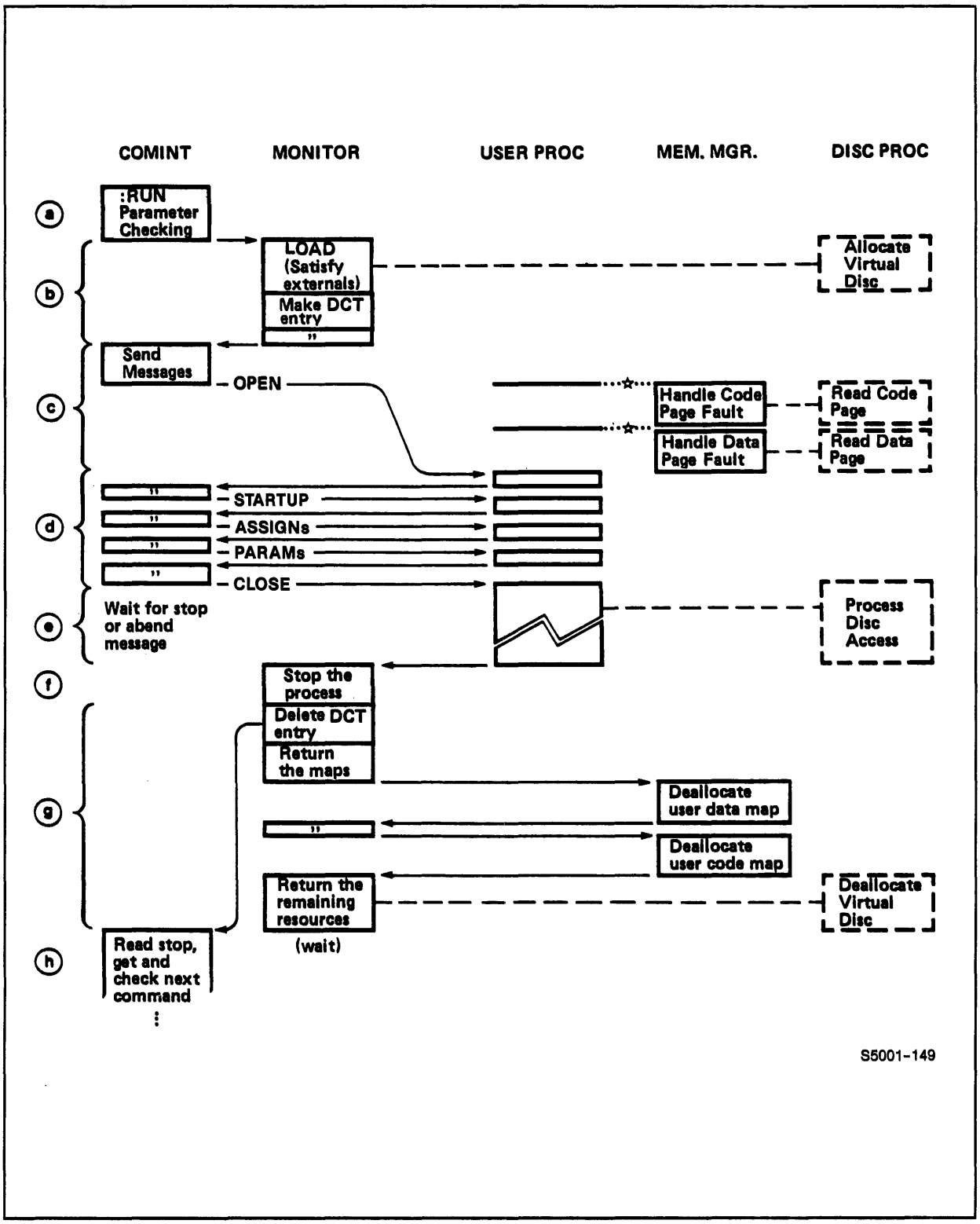


Figure 11-18. Process Life Cycle

a. When the command interpreter reads a valid RUN command to execute the user process, it calls the NEWPROCESS procedure to create the process. NEWPROCESS sends a message to the appropriate monitor (in the CPU where the process is to run) requesting the start of the new process.

   The command interpreter, now within the NEWPROCESS procedure, awaits a reply to its Newprocess message to determine if the request was successful.

b. The monitor that receives the message initializes and starts a prototype process. This process opens the code and library files, and if fixup (linking the code and library external references to the system procedures) is necessary, this is also done at this time. Fixup consists of searching the System Entry Point Table to satisfy all such external references and changing the calls in the program file appropriately. Once the code from a program file has been run, it may be run repeatedly without satisfying the external references again. But if there is any possibility that the GUARDIAN operating system has been changed and the location of its externals affected, all external references in the program file must be resolved again when the program is next run.

   Once the prototype process has performed fixup and other operations, its state is changed to begin executing the main procedure of the program. The monitor opens the code files as read-only swap files, and creates and opens the data segment swap file as a read/write file. If the process is named, it is allocated an entry in the Destination Control Table.

   When complete, with or without errors, the monitor replies to the originator of the new-process message (command interpreter, in this case) with an error code, or zeros for no error.

c. The command interpreter determines that the user process is accepting messages, and sends an Open message to it. This message informs the user process that another process has opened it as a file. The incoming message is queued on the user process's PCB and the command interpreter suspends, awaiting a response.

   Now the Dispatcher selects the user process as the next process to execute. The Dispatcher sets the CPU registers to the values in the process's PCB and transfers control to the process by exiting the interrupt environment.

A program usually begins execution with none of its code or data pages in main memory. Each time it references an absent page, the program is suspended until the memory manager can fetch the page from disc. Initially, a program causes a flurry of page faults, but quickly obtains enough pages to execute, with only occasional faults caused by accessing seldom used procedures or data structures.

d. The user process reads the open message queued on its PCB and sends a reply to the command interpreter.

   In response, the command interpreter awakens and sends a start-up message to the user process. This message contains the parameters and other information supplied in the RUN command. In this case, the user process replies to this message with a special return value which indicates that the user process's logic is prepared to receive any additional messages that the command interpreter might have to send.

   When requested by the user process, the command interpreter sends any information specified in ASSIGN or PARAM commands that were entered by the user. When all such messages have been exchanged and acknowledged, the command interpreter sends a close message to the user process. If the user process was not run with the nowait option, the command interpreter waits until it receives a message from the operating system that the user process has terminated before prompting for another command.

e. The user process continues execution. This execution, however, might be interrupted by suspensions for input-output activity, or by execution of higher-priority processes. (Remember that the CPU is shared by all processes and interrupt handlers, and executes only one instruction at a time. Each time a new process executes, the state of the previous process is saved and the CPU is then reset to reflect the code and data environment of the process selected for execution.)

f. When the user process has completed its operations, it calls the STOP procedure to terminate itself. This call may be one that the user has coded explicitly, or one that the compiler provided at the logical end of the program. The STOP procedure sends a message to the monitor requesting user process termination.

g. In response to the message from the STOP procedure, the monitor now stops the process by breaking its communication

links and returning the resources it was using. The monitor also closes all files that were opened for this process, and removes the process's entry in the DCT.

The monitor then requests the memory manager to deallocate all physical pages held by the process. The memory manager responds by deallocating the pages held by this process, making them available for other processes or the system to use.

Then, the monitor sends a message to the disc process to close any swap files belonging to the user process, deletes any messages queued on the terminating process's PCB, and returns the PCB. When all the user process's resources have been returned to the system, the monitor sends a Stop message to the process's ancestor, which in this case is the command interpreter. The monitor then awaits another incoming request.

h. The command interpreter is now able to resume execution and read the termination message sent to it by the monitor process. It checks this message and determines that the last user process that it created has terminated. Since this process was created in a waited manner, its termination permits the command interpreter to prompt for the next command.

PROCESS PAIRS

Fault-tolerant operation of the NonStop system depends upon the concept of process pairs, where primary and backup processes form redundant sets that promote fault-tolerance. These process pairs may be employed by both the operating system and its users; however, they are implemented in different ways, depending upon which entity creates them.

Process pairs usually are named; the naming convention makes them easier to work with. The process name as well as the CPU's and PINs of the primary and backup processes are recorded in the Destination Control Table (DCT) (Figure 11-19). Thus, when a user wants to communicate with a named process pair, the operating system locates the process by looking up its name in the DCT and determining the associated PID (<cpu>,<pin>) identifier.

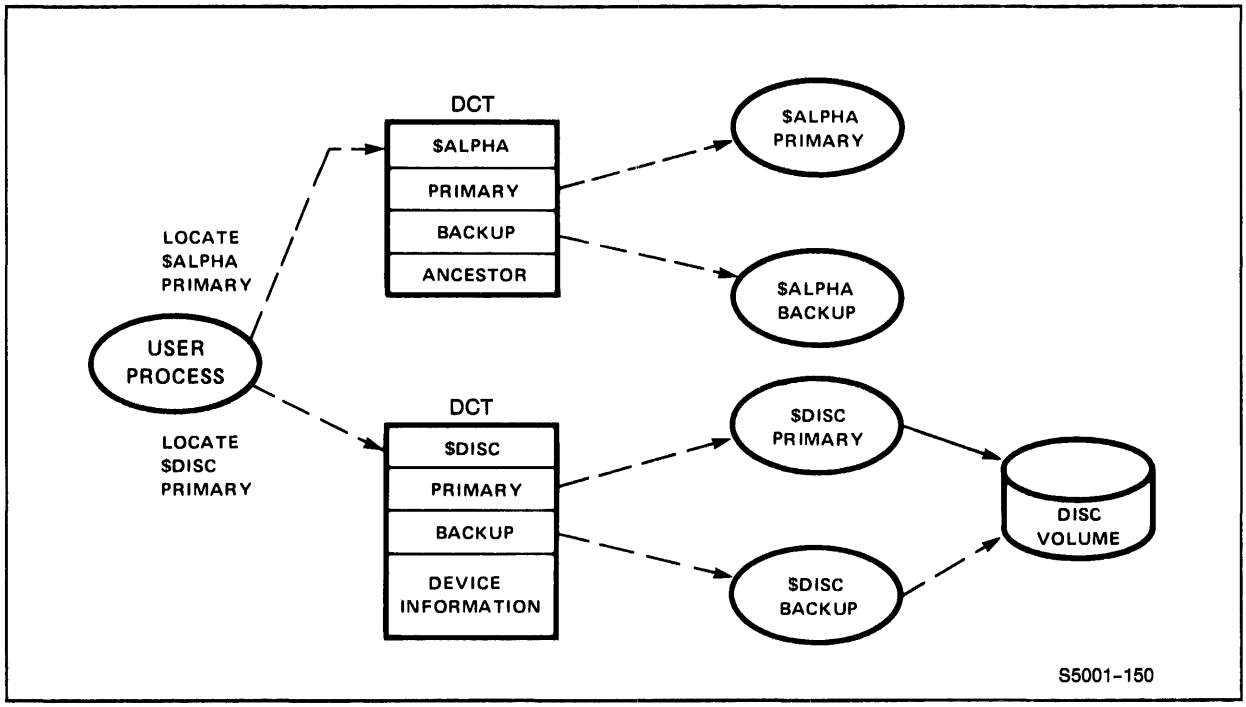


Figure 11-19. Named Process Pair Versus Named Device

To provide fault-tolerance at the system level, each input-output device is controlled by a process pair. When an application program opens (or otherwise wishes to access) an I/O device, the file system first finds the device's entry in the Destination Control Table. This table (DCT) contains the CPU and process numbers for the device's primary and backup processes. The file system then transmits the user's request in a message to the device's primary process.

Under normal circumstances, user processes communicate with both named and input-output process pairs through the file system. As far as the user is concerned, the process name or device identifier represents a single active process. In actuality, however, the name/identifier references both the primary and backup, with the backup member remaining dormant (except for processing Checkpoint messages) until the primary fails. The file system remains responsible for directing messages to the appropriate member of the pair.

For instance, suppose a user opens and writes data to a process that he identifies as $ALPHA (Figure 11-20). To keep its backup informed of current requests, $ALPHA's primary process sends checkpoints to the backup. Now suppose, at some later time, the CPU on which the primary is running fails. The file system once

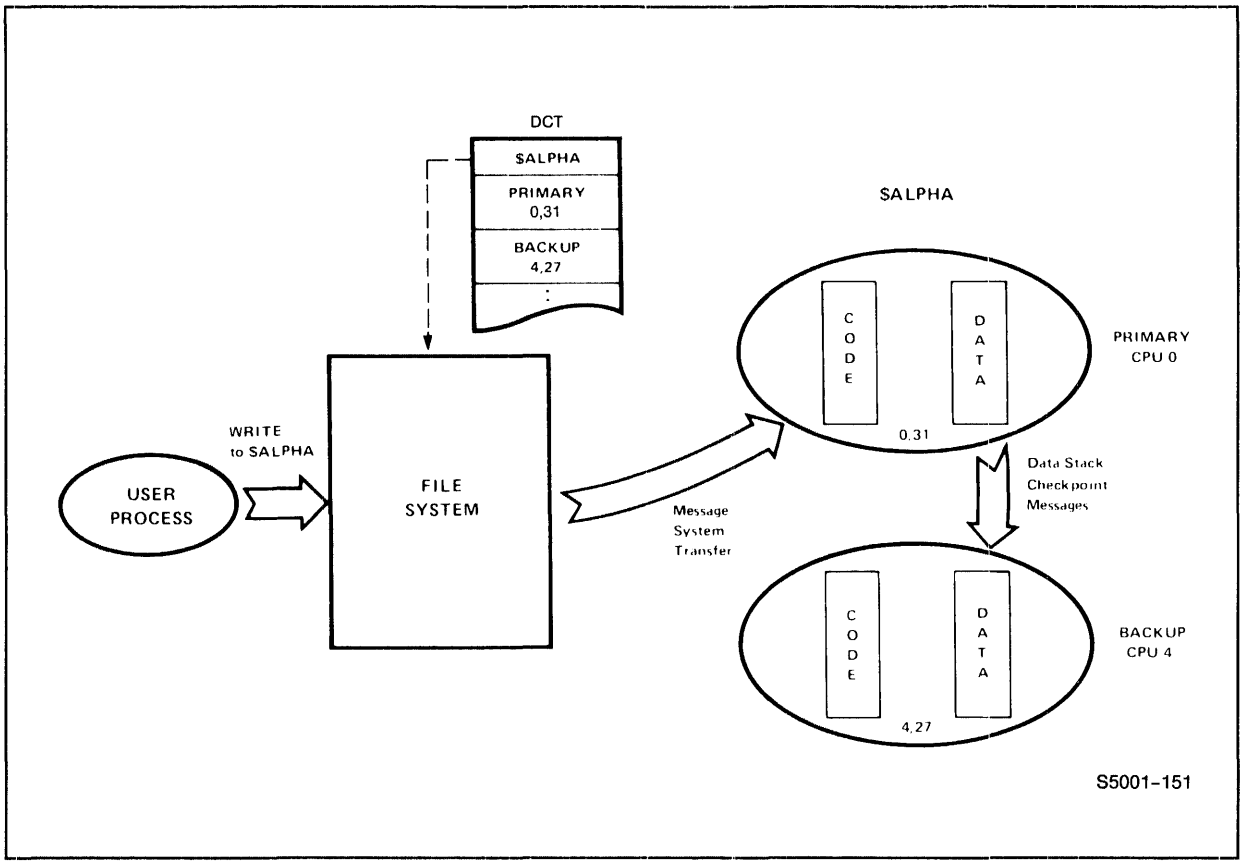


Figure 11-20. Process Pair Backup

again tries to reference $ALPHA's failed primary. The file system always routes data to the primary member of the process pair (Figure 11-21, Event A). When such a transfer fails, any outstanding messages to the process are cancelled and a "path error" to the device or process results. In this case, the backup process becomes the primary (Event B) and the operation may be retried to the new primary (Event C).

Input/output processes operate in a parallel way. When a message request for the device occurs, the operating system sends the message to the first process (primary) in the DCT entry for the device. If the message cannot be delivered, the error indication causes the operating system to switch the primary and backup entries in the DCT. When the device is a disc, and the syncdepth is greater than zero, the system resends the request to the new primary process. The operating system handles this error recovery automatically, and the user remains unconcerned with what process actually handled the request. If the device is not a disc, such a failure results in an error without retry.

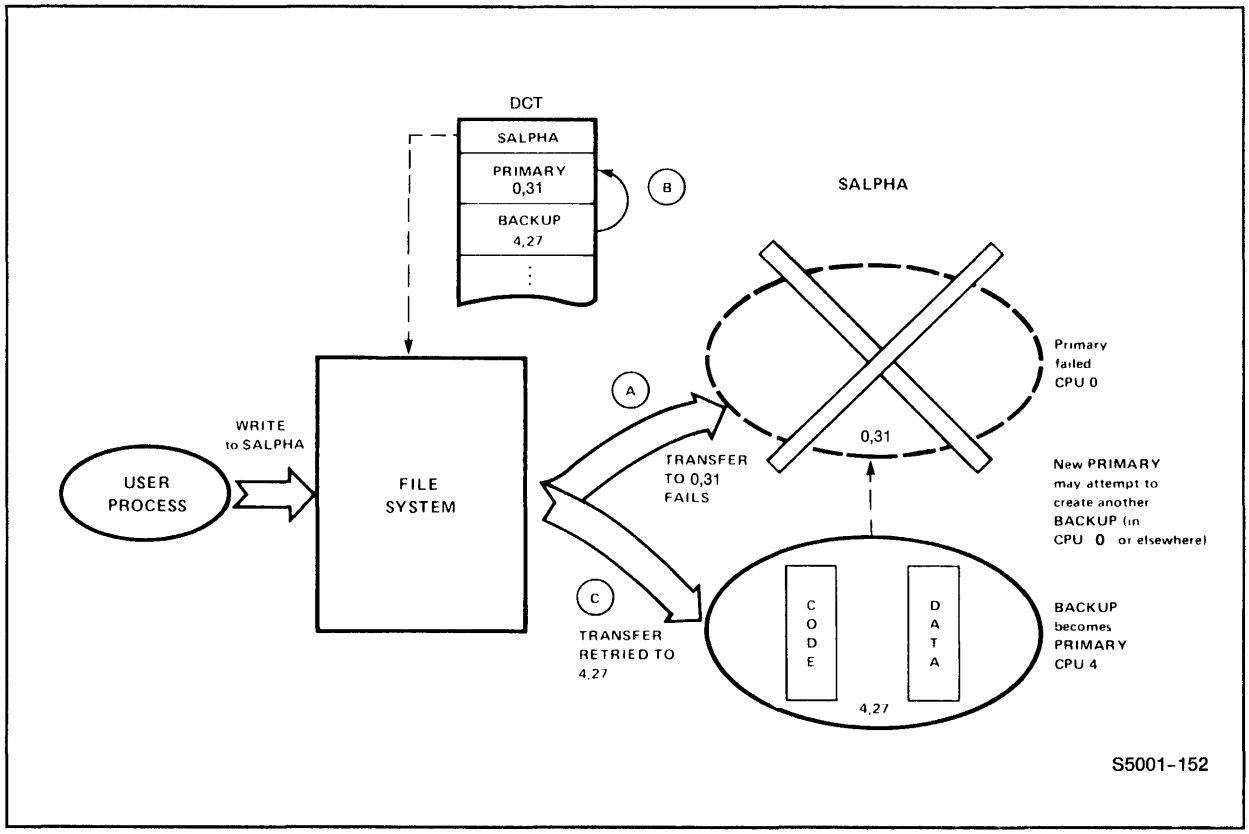


Figure 11-21. Primary Process Failure

REQUESTER/SERVER RELATIONSHIPS

In the case of process pairs, described above, each process is the functional equivalent of the other. Beyond this, however, two processes can be designed to be responsible for entirely different functions. For instance, a user might easily divide an application into two processes so that:

- One process handles the front-end terminal interface.
- The other process manages all data base and other disc accesses.

Then, if fault-tolerant performance is important, the user might create each of these processes as a process pair.

The relationship between two such communicating processes (or pairs of processes) is defined by their functions, and is called a requester-server relationship. The requester initiates a

request to access a resource that is logically "owned" by the server; the server performs the requested action and replies to the requester.

In the above example, the requester process (terminal front-end handler, $REQ) opens the server process (data-base accessor, $SER) as a file. Incoming commands from the terminal cause the requester process to call a system library procedure named WRITEREAD (Figure 11-22). This procedure not only sends a request message to the server, but also expects a response from the server. The server, in turn, opens a file named $RECEIVE and calls the system library procedure READUPDATE. This procedure not only reads from $RECEIVE the message sent by the requester but also permits the server to send a response to the requester. After the server has read the request message, it performs the required operations. (These typically include data base accesses.) The server then sends its response, and completes the transaction by calling the system library procedure REPLY. This combination of WRITEREAD, READUPDATE, and REPLY procedures allows a two-way data transfer within the framework of a single message and illustrates the way in which requesters and servers are interlocked. That is, once a request has been made, the requester typically cannot make further requests until the server replies.

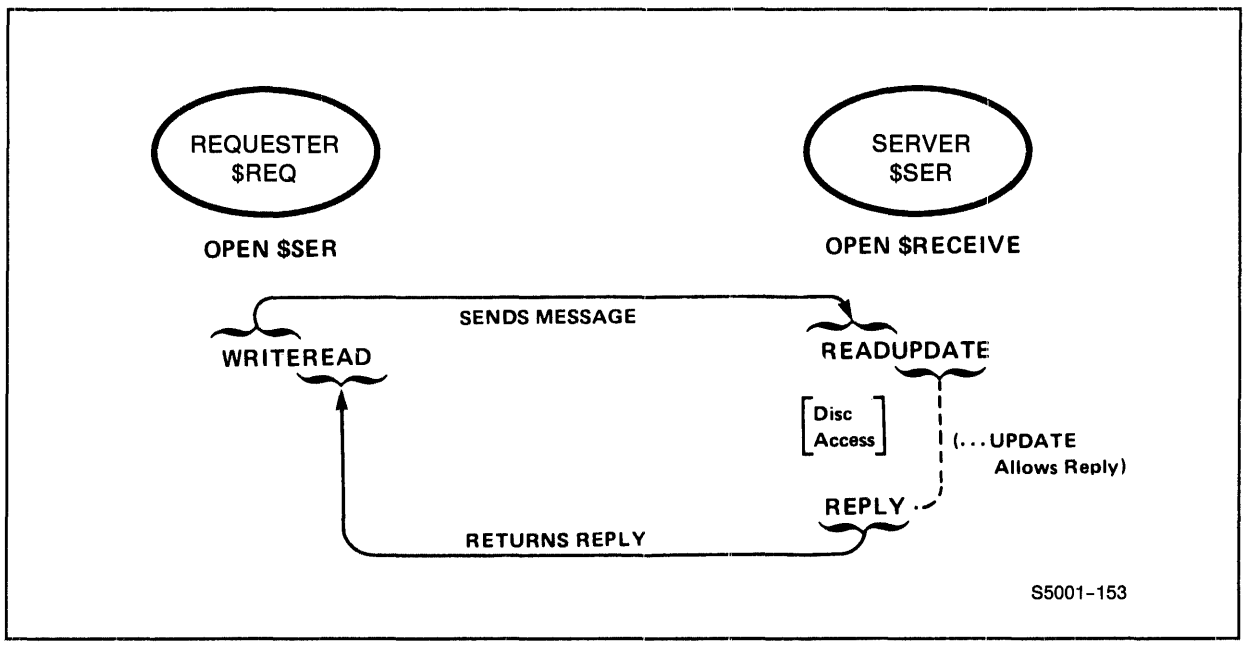


Figure 11-22. Requester-Server Pair

In this way, the requester always remains the controlling process in the relationship. The server, on the other hand, is mainly a passive process, awaiting messages from the requester and acting only when those requests arrive. Under typical circumstances, when a server finishes processing all incoming requests, it suspends and awaits further messages.

A typical example of request handling appears below in Steps a through c:

a. The requester opens the server process as a file and then sends messages to it. The requester identifies the server by:

    1. The server's process ID, if the server is an unnamed process.

    2. The server's process name, if the server is actually a named process or process pair.

b. After the server completes any required initialization, it opens and reads the $RECEIVE file. (This file must be opened and read in order to pick up any incoming messages.) Multithreaded servers open $RECEIVE with a receive depth parameter greater than zero in the OPEN call; this permits the server to send replies to the requester. $RECEIVE acts as a funnel through which all incoming requests arrive. The server reads the requester's message from $RECEIVE, and if a backup server exists, may also checkpoint the request to it.

    By reading $RECEIVE, the server simply reads unsolicited messages already transmitted to that file by other processes. Thus, while the requester must know the identity of the server to send a message, the server need only pick up messages from $RECEIVE. The message and file systems keep track of the identity of the requester. For this reason, the server only needs to reply to such incoming messages and the file system automatically directs the replies back to the appropriate requester.

c. The server reads the requester's message from $RECEIVE, interprets the action required, and performs the requested function. Typically, the function involves returning information to the requester by replying to that process's message.

Some additional techniques are available to assist users in developing more sophisticated requester-server applications. These techniques are supported by the following file-open options:

- nowait depth (Bits 12 through 15 of the <flags> parameter in the OPEN procedure call.)
- sync depth (Used by requester process)
- receive depth (Used by server process)

To clarify how these options may be used, the following discussion illustrates their application in the requester and server environments.

Requester Environment. The concept of nowait input-output transfers applies to files used by both requesters and servers. Simply stated, nowait input-output allows a process to begin a transfer and then continue execution in parallel with it.

Under normal circumstances (wait input-output), the user process suspends until the input-output completes. In such cases, the process is assured that the transfer has been completed before the process resumes execution. When a nowait input-output transfer is requested, on the other hand, the process remains in execution and must check for the completion of the input-output by calling the system library procedure AWAITIO.

The nowait input-output facility is requested in the call to the OPEN procedure. The caller specifies a nowait depth when the requester process opens a server process as a file. If the requester specifies a non-zero value for nowait depth, this value limits the number of outstanding requests that may be queued against the server process at any one time. For instance, with a nowait depth of three, no more than three data transfers to the server could be outstanding at any instant--that is, one of the three requests must be completed before another can be successfully queued. This queueing can be used to permit the requester and server to operate asynchronously. The requester can fill the server's input queue as needs arise, and the server can respond to requests in the queue as time permits.

Another facility available to both requester and server processes is controlled by the sync depth parameter in the requester's call to open a server process pair as a file. This facility enables the requester and server to coordinate their communications and is completely independent of the nowait depth value. A nonzero sync depth value has two effects:

1. It causes the file system to automatically retry requests which were unsuccessfully sent to the server. (The retry is directed to the backup process of the server process pair.)

2. It causes the sync depth value to be sent to the server in the OPEN message that notifies the server process that it has been opened by the requester. Once the server receives the sync depth value, it expects that all incoming messages will contain a sync ID value. This information should enable the server to associate status returns with sync ID values, and to use logic to detect and ignore duplicate requests. The number of messages sent to the server between checkpoints in the requester should not exceed the sync depth value.

NOTE

With the exception of the automatic retry done by the file system, all the logic described above must be provided by either the application programmer or the high-level language (such as FORTRAN or COBOL) in which he is working.

Server Environment. In order to receive messages from the requester, the server process must open the $RECEIVE file. Nowait depth has a very limited application in this environment--its maximum value is only one. The value of nowait depth has the following effect when no incoming requests are pending on $RECEIVE:

- A nowait depth of zero causes the server process to wait for an incoming request on $RECEIVE.
- A nowait depth of one allows the server to post one nowait READ operation against $RECEIVE to monitor this file for incoming requests while the server is otherwise occupied. As before, the completion of any nowait request must be determined by calling the system library procedure AWAITIO.

Note, however, that since $RECEIVE is a single file, a single READ operation posted against this file is all that is needed to monitor it for input.

Another facility available to the server is defined by the receive depth parameter specified in the OPEN call for the $RECEIVE file. Receive depth specifies the number of requests that may be read before any reply is returned to a requester. Receive depth must be greater than or equal to 1, to enable the server process to return a reply to the requester. In other words, receive depth specifies the number of READUPDATES that can be issued before performing a reply. For instance:

receive depth	Number of READUPDATEs Allowed Before Performing REPLY Operation
0	0 (No READUPDATE allowed--file can only use READ.)
1	1
2	2
.	.
.	.
.	.

The receive depth facility is completely independent of the nowait depth discussed earlier, and allows the server to examine its input queue, make decisions about the order in which to service requests, and respond to the requests in arbitrary order.

The options discussed above may be used singly or in combination in the requester-server process environment. While their use was not required in the above example, they may be very helpful tools in implementing applications where more sophisticated queueing and error recovery is necessary.

Multiple Requester-Server Relationships. The previous examples have covered single requester-server applications. But in more complex relationships, $RECEIVE can function as a universal two-way communication path among multiple requesters and servers. In fact, many servers, each with its own $RECEIVE file, can receive input from more than one requester and any requester can communicate with more than one server (Figure 11-23).

The requester-server concept indicates a relationship between processes, not the exclusive duty of a process. In fact, some processes perform both functions in the course of their execution. This idea is the basis of "pass-through" arrangements, where a requester transmits a request to a server and this server, in turn, also functions as a requester, transmitting the request to another server (Figure 11-24). In these arrangements, each process is frequently a member of a process pair, with a primary and backup process involved at each point in the communication stream.

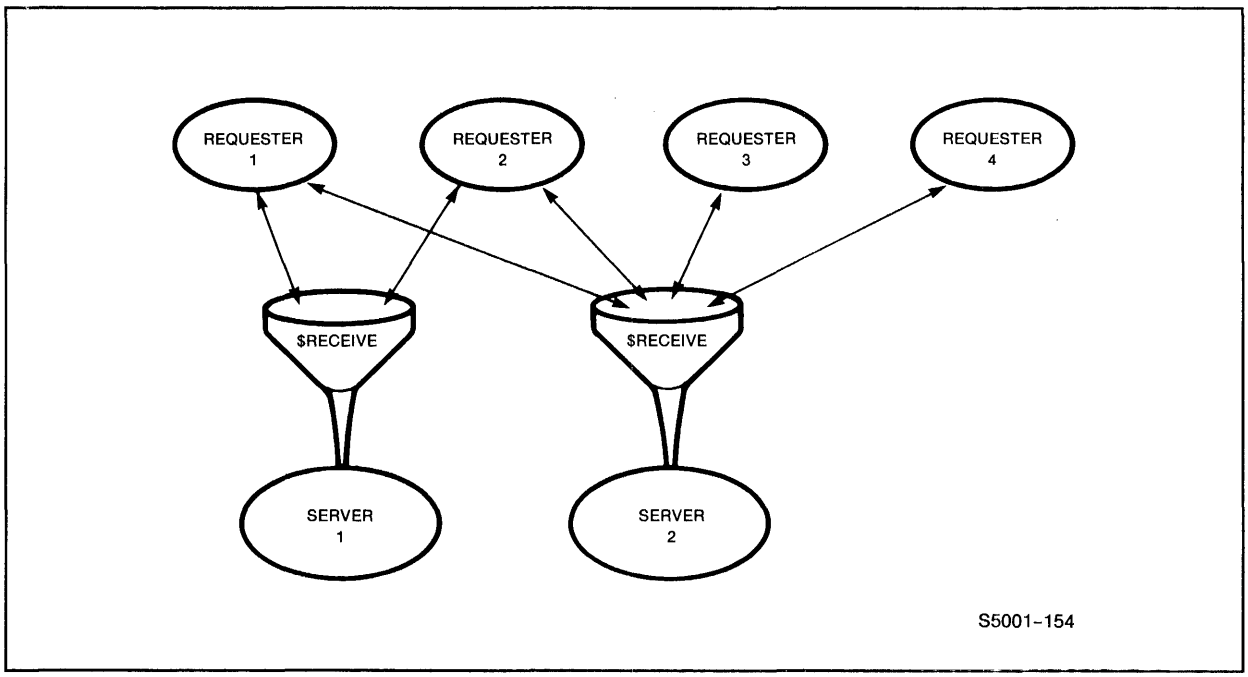


Figure 11-23. Multiple Requester-Server Relationships

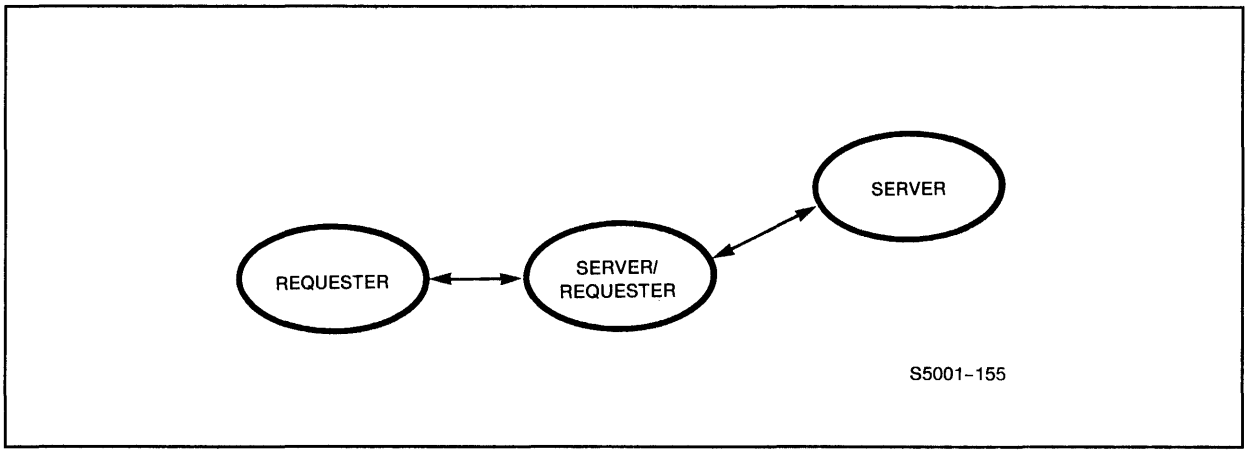


Figure 11-24. Pass-Through Arrangement

Although system processes do not communicate with each other by using process files and $RECEIVE, many of the functional relationships that the operating system depends upon are essentially requester-server pairs. Communication between these processes takes place at the message system level--but is still functionally equivalent to the way that application requester/servers communicate through the file system.

As an example of system process intercommunication, consider the case where an ordinary user process calls file system procedures to read and write information to a terminal, sending requests to the terminal process that controls the device (Figure 11-25). In this case, the user process is the requester and the terminal process is the server. In a similar way, when the user process calls file system procedures to read or write data to a disc file, it sends messages to the disc process controlling the device. Again, the user process is the requester but the disc process is now the server. In this way, input-output processes (and other system processes such as the monitor and operator processes) depend on requester-server relationships.

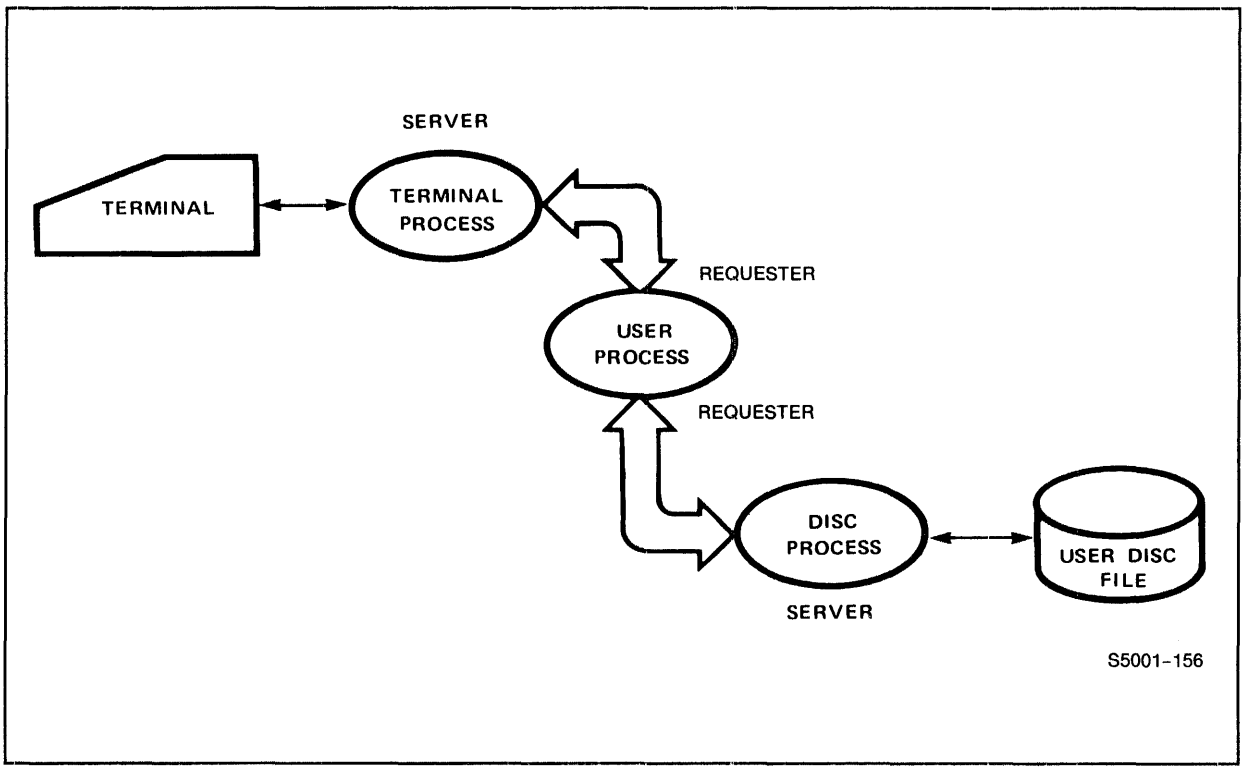


Figure 11-25. Communication with System Processes

To illustrate requester-server relations within an application, consider a transaction processing system. In this system, requester components might handle terminal input-output, validate input fields for data consistency, convert this data to internal format, and control transaction flow. Server components, on the other hand, might perform such functions as:

a. Reading messages from a requester

b. Reacting to this request by reading, writing to, updating, or deleting information from the data base

c. Building a reply containing data from the data base or control information describing an error that occurred

d. Transmitting a reply message to the requester

As another application example, suppose that a transaction processing system performs three main functions: checking credit, adding a new order, and updating an existing order. (Each of these functions is handled by a corresponding server process.) The requester process reads information from the user's terminal, constructs a message, and sends the message to an appropriate server process based on the request. The three servers are responsible for all activity on the data base (Figure 11-26).

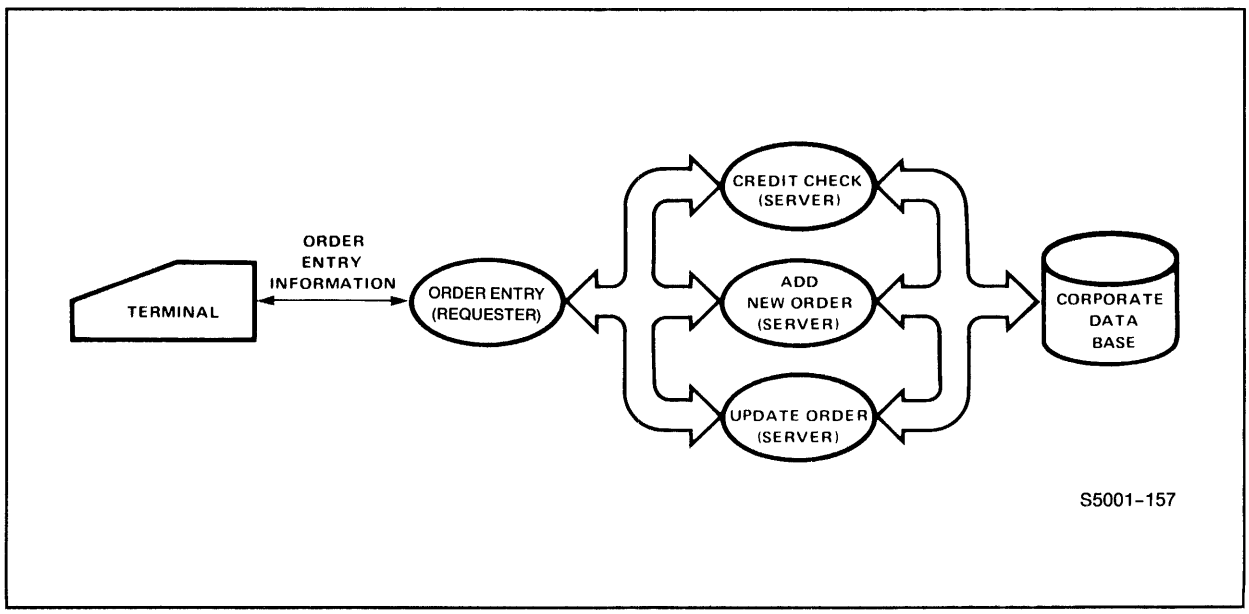


Figure 11-26. Communication with Application Processes

A more complex example appears in Figure 11-27. In this example, multiple requester processes share access to multiple server processes. In the case of the credit-check function, multiple copies of the same server were created in different CPUs to increase performance and throughput.

A main advantage of requester-server relationships is modularity--the ability to implement a system in discrete modules or components that can work in parallel. Thus multiple requesters and multiple servers can work together to accomplish a single application. Modularity also permits smaller, more manageable components that are easy to define, write, debug, integrate into the system, and maintain. It makes a system more flexible by letting designers easily add new user functions that employ services already provided by existing application processes. And finally, it permits system expansion by allowing a flexible distribution of terminals, requesters, and servers among a system's CPU's--and perhaps even among systems in a network. All of these factors make it easier to optimize application throughput and performance.

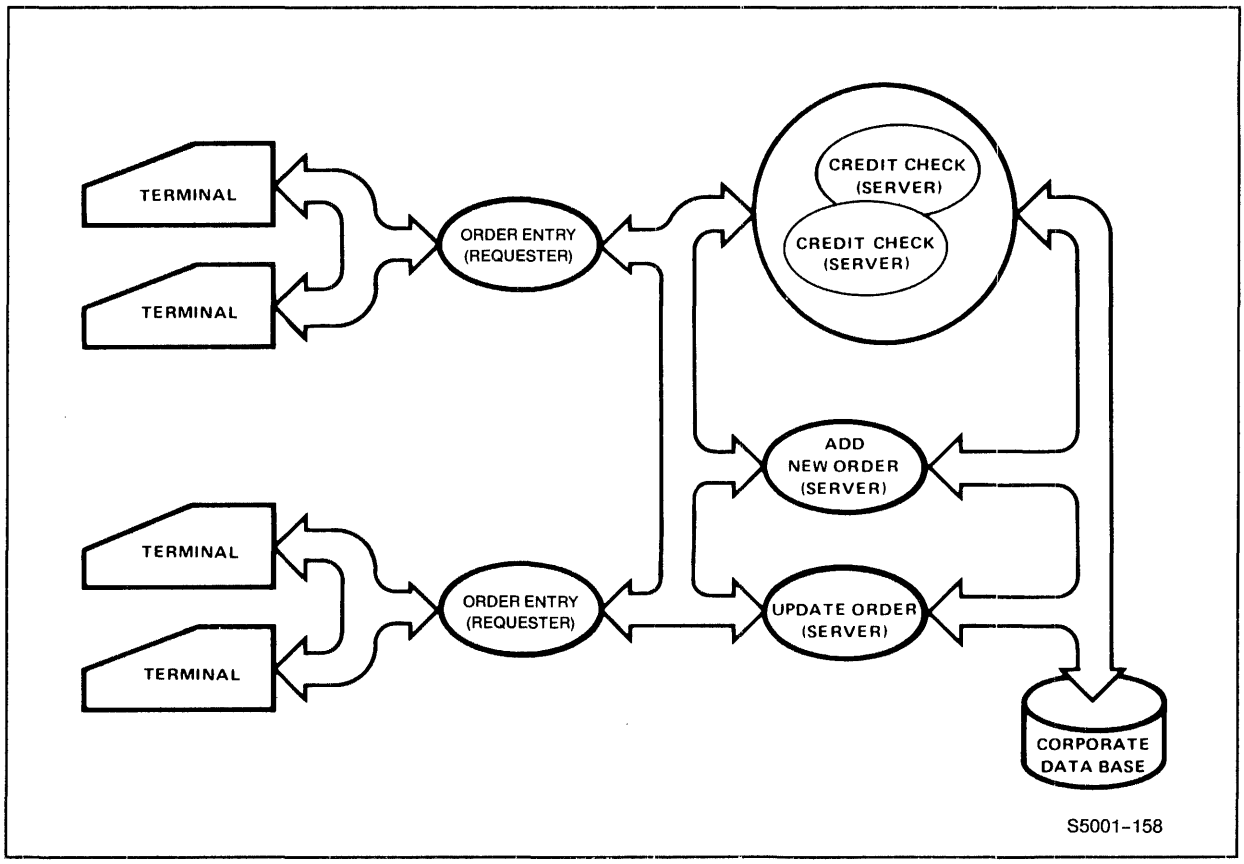


Figure 11-27. Application with Multiple Requesters and Servers

APPENDIX A

HARDWARE INSTRUCTION LISTS

This appendix provides a number of reference tables pertaining to the instruction sets of the NonStop II and NonStop TXP processors.

The first two tables list all instructions in the instruction set with their mnemonics and opcodes, first in alphabetical order and then grouped by type of instruction. The remaining tables provide binary coding details for most of the instructions, grouped according to the coding patterns of the fields of the instruction words. (For example, all memory reference instructions are listed together.) These tables break down each instruction, bit by bit, into its component parts, indicate the operands, results, and ENV Register bit settings, and show relationships between similar instructions.

The following tables are included in this appendix:

A-1. Alphabetical List of Instructions
A-2. Categorized List of Instructions
A-3. Binary Coding, Memory Reference Instructions
A-4. Binary Coding, Immediate Instructions
A-5. Binary Coding, Move/Shift/Call/Extended Instructions
A-6. Binary Coding, Branch Instructions
A-7. Binary Coding, Stack Instructions
A-8. Binary Coding, Decimal Arithmetic Instructions
A-9. Binary Coding, Floating-Point Instructions

A key at the end of each table explains the symbols used. For some instructions, the six-digit opcode notation used in Tables A-1 and A-2 cannot give complete information about the opcode. For instance, the distinctions between QUP and QDWN, ORRI and ORLI, and LWP and LBP cannot be clearly shown. For complete information, refer to the entries for these instructions in Tables A-3 through A-9.

Table A-1. Alphabetical List of Instructions

Mnemonic	Description	Octal Code	
ADAR	Add A to Register	00016-	
ADDI	Add Immediate	104---	
ADDS	Add to S	002---	
ADM	Add to Memory	-74---	
ADRA	Add Register to A	00014-	
ADXI	Add to Index Immediate	104---	
ALS	Arithmetic Left Shift	0302--	
ANG	AND to Memory	000044	
ANLI	AND Left Immediate	007---	
ANRI	AND Right Immediate	006---	
ANS	AND to SG Memory	000034	
ANX	AND to Extended Memory	000046	
ARS	Arithmetic Right Shift	0303--	
ASPT	Address of Segment Page Table Header	000470	*
BANZ	Branch on A	-154--	
BAZ	Branch on A Zero	-144--	
BCLD	Bus Cold Load	000452	*
BEQL	Branch if Equal	-12---	
BFI	Branch Forward Indirect	000030	
BGEQ	Branch if Greater or Equal	-13---	
BGTR	Branch if Greater	-11---	
BIC	Branch if Carry	-10---	
BIKE	Bicycle While Idle	000464	*
BLEQ	Branch if Less or Equal	-16---	
BLSS	Branch if Less	-14---	
BNDW	Bounds Test Words	000450	*
BNEQ	Branch if Not Equal	-15---	
BNOC	Branch if No Carry	-17---	
BNOV	Branch if No Overflow	-164--	
BOX	Branch on X	-1-4--	
BPT	Instruction Breakpoint Trap	000451	
BSUB	Branch to Subprocedure	-174--	
BTST	Byte Test	000007	
BUN	Branch	-104--	
CAQ	Convert ASCII to Quad	000262	$
CAQV	Convert ASCII to Quad with Initial Value	000261	$
CCE	Condition Code Equal to	000016	
CCG	Condition Code Greater than	000017	
CCL	Condition Code Less than	000015	
CDE	Convert Doubleword to Extended Float	000334	#
CDF	Convert Doubleword to Float	000306	#
CDFR	Convert Doubleword to Float (Round)	000326	#

Table A-1. Alphabetical List of Instructions (Continued)

Mnemonic	Description	Octal Code	
CDG	Count Duplicate Words	000366	
CDI	Convert Doubleword to Integer	000307	
CDQ	Convert Doubleword to Quad	000265	$
CDX	Count Duplicate Words Extended	000356	
CED	Extended Float to Doubleword	000314	#
CEDR	Extended Float to Doubleword (Round)	000315	#
CEF	Extended Float to Float	000276	#
CEFR	Extended Float to Float (Round)	000277	#
CEI	Extended Float to Integer	000337	#
CEIR	Extended Float to Integer (Round)	000316	#
CEQ	Extended Float to Quadrupleword	000322	#
CEQR	Extended Float to Quadrupleword (Round)	000323	#
CFD	Floating to Doubleword	000312	#
CFDR	Floating to Doubleword (Round)	000313	#
CFE	Floating to Extended Float	000325	#
CFI	Floating to Integer	000311	#
CFIR	Floating to Integer (Round)	000310	#
CFQ	Floating to Quadrupleword	000320	#
CFQR	Floating to Quadrupleword (Round)	000321	#
CID	Convert Integer to Doubleword	000327	
CIE	Convert Integer to Extended Float	000332	#
CIF	Convert Integer to Floating	000331	#
CIQ	Convert Integer to Quad	000266	$
CLQ	Convert Logical to Quad	000267	$
CMBX	Compare Bytes Extended	000422	
CMPI	Compare Immediate	001---	
CMRW	Correctable Memory Error Read/Write	000432	*
COMB	Compare Bytes	1262--	
COMW	Compare Words	0262--	
CQA	Convert Quad to ASCII	000260	$
CQD	Convert Quad to Doubleword	000247	$
CQE	Convert Quad to Extended	000336	#
CQER	Convert Quad to Extended (Round)	000335	#
CQF	Convert Quad to Floating	000324	#
CQFR	Convert Quad to Floating (Round)	000330	#
CQI	Convert Quad to Integer	000264	$
CQL	Convert Quad to Logical	000246	$
CRAX	Convert Relative to Absolute Extended	000423	*
DADD	Double Add	000220	
DALS	Double Arithmetic Left Shift	1302--	
DARS	Double Arithmetic Right Shift	1303--	
DCMP	Double Compare	000225	

Table A-1. Alphabetical List of Instructions (Continued)

Mnemonic	Description	Octal Code	
DDIV	Double Divide	000223	
DDTX	DDT Request (NonStop TXP processor only).	000456	*
DDUP	Double Duplicate	000006	
DFG	Deposit Field in Memory	000367	
DFS	Deposit Field in System	000357	
DFX	Deposit Field in Extended Memory	000416	
DISP	Dispatch	000073	*
DLEN	Disc Record Length	000070	@
DLLS	Double Logical Left Shift	1300--	
DLRS	Double Logical Right Shift	1301--	
DLTE	Delete Element from List	000054	*
DMPY	Double Multiply	000222	
DNEG	Double Negate	000224	
DOFS	Disc Record Offset	000057	@
DPCL	Dynamic Procedure Call	000032	
DPF	Deposit Field	000014	
DSUB	Double Subtract	000221	
DTL	Determine Time Left for Element	000207	*
DTST	Double Test	000031	
DXCH	Double Exchange	000005	
DXIT	DEBUG Exit	000072	*
EADD	Extended Floating-Point Add	000300	#
ECMP	Extended Floating-Point Compare	000305	#
EDIV	Extended Floating-Point Divide	000303	#
EIO	Execute I/O	000060	*
EMPY	Extended Floating-Point Multiply	000302	#
ENEG	Extended Floating-Point Negate	000304	#
ESUB	Extended Floating-Point Subtract	000301	#
EXCH	Exchange	000004	
EXIT	Exit Procedure	125---	
FADD	Floating-Point Add	000270	#
FCMP	Floating-Point Compare	000275	#
FDIV	Floating-Point Divide	000273	#
FMPY	Floating-Point Multiply	000272	#
FNEG	Floating-Point Negate	000274	#
FRST	Firmware Reset	000405	*
FSUB	Floating-Point Subtract	000271	#
FTL	Find Position in Time List	000206	*
HALT	Processor Halt	000074	*
HIIO	High-Priority Interrogate I/O	000062	*
IADD	Integer Add	000210	
ICMP	Integer Compare	000215	

Table A-1. Alphabetical List of Instructions (Continued)

Mnemonic	Description	Octal Code	
IDIV	Integer Divide	000213	
IDX1	Calculate Index, 1 Dimension	000344	#
IDX2	Calculate Index, 2 Dimension	000345	#
IDX3	Calculate Index, 3 Dimension	000346	#
IDXD	Calculate Index, Bounds in Data Space	000317	#
IDXP	Calculate Index, Bounds in Code Space	000347	#
IIO	Interrogate I/O	000061	*
IMPY	Integer Multiply	000212	
INEG	Integer Negate	000214	
INSR	Insert Element into List	000055	*
ISUB	Integer Subtract	000211	
IXIT	Interrupt Exit	000071	*
LADD	Logical Add	000200	
LADI	Logical Add Immediate	003---	
LADR	Load Address	-7----	
LAND	Logical AND	000010	
LBA	Load Byte via A	000364	
LBAS	Load Byte via A from System	000354	
LBP	Load Byte from Program	-2-4--	
LBX	Load Byte Extended	000406 0266--	
LBXX	Load Byte Extended, Indexed	0256--,	
LCKX	Lock Down Extended Memory	000430	*
LCMP	Logical Compare	000205	
LDA	Load Double via A	000362	
LDAS	Load Double via A from System	000352	
LDB	Load Byte	-5----	
LDD	Load Double	-6----	
LDDX	Load Double Extended	000412	
LDI	Load Immediate	100---	
LDIV	Logical Divide	000203	
LDLI	Load Left Immediate	005---	
LDRA	Load Register to A	00013-	
LDX	Load X	-3----	
LDXI	Load X Immediate	10----	
LIOC	Load IOC	000457	*
LLS	Logical Left Shift	0300--	
LMPY	Logical Multiply	000202	
LNEG	Logical Negate	000204	
LOAD	Load	-4----	
LOR	Logical OR	000011	
LQAS	Load Quadrupleword via A from SG	000445	*

Table A-1. Alphabetical List of Instructions (Continued)

Mnemonic	Description	Octal Code	
LQX	Load Quadrupleword Extended	000414	
LRS	Logical Right Shift	0301--	
LSUB	Logical Subtract	000201	
LWA	Load Word via A	000360	
LWAS	Load Word via A from System	000350	
LWP	Load Word from Program	-2----	
LWUC	Load Word from User Code Space	000342	
LWX	Load Word Extended	000410	
LWXX	Load Word Extended, Indexed	0254--, 0264--	
MAPS	Map In a Segment	000042	*
MBXR	Move Bytes Extended, Reverse	000420	
MBXX	Move Bytes Extended, and Checksum	000421	
MNDX	Move Words while Not Duplicate, Extended.	000227	
MNGG	Move Words while Not Duplicate	000226	
MOND	Minus One Double	000001	
MOVB	Move Bytes	126---	
MOVW	Move Words	026---	
MRL	Merge onto Ready List	000075	*
MVBX	Move Bytes Extended	000417	
MXFF	Mutual Exclusion Off	000041	*
MXON	Mutual Exclusion On	000040	*
NOP	No Operation	000000	
NOT	Not	000013	
NSAR	Nondestructive Store A in a Register	00012-	
NSTO	Nondestructive Store	-34---	
ONED	One Double	000003	
ORG	OR to Memory	000045	
ORLI	OR Left Immediate	0044--	
ORRI	OR Right Immediate	004---	
ORS	OR to SG Memory	000035	
ORX	OR to Extended Memory	000047	
PCAL	Procedure Call	027---	
POP	Pop from Stack	124---	
PSEM	"P" a Semaphore	000076	*
PUSH	Push to Stack	024---	
QADD	Quad Add	000240	
QCMP	Quad Compare	000245	$
QDIV	Quad Divide	000243	$
QDWN	Quad Scale Down	00025-	
QLD	Quad Load	00023-	
QMPY	Quad Multiply	000242	$

Table A-1. Alphabetical List of Instructions (Continued)

Mnemonic	Description	Octal Code	
QNEG	Quad Negate	000244	$
QRND	Quad Round	000263	$
QST	Quad Store	00023-	
QSUB	Quad Subtract	000241	
QUP	Quad Scale Up	00025-	
RCHN	Reset I/O Channel	000447	*
RCLK	Read Clock	000050	
RCPU	Read Processor Number	000051	
RDE	Read E Register	000024	
RDP	Read P Register	000025	
RIBA	Read INTA and INTB Registers	000440	*
RIR	Reset Interrupt	000063	*
RMAP	Read Map (NonStop II processor only)	000066	*
RPT	Read Process Timer	000442	*
RPV	Read PROM Version Numbers	000216	*
RSMT	Read from Operations & Service Processor.	000436	*
RSPT	Read Segment Page Table Entry	000424	*
RSUB	Return from Subprocedure	025---	
RSW	Read Switches	000026	
RUS	Read Micro State	000461	*
RWCS	Read LCS	000402	*
RXBL	Read Extended Base and Limit	000426	*
SBA	Store Byte via A	000365	
SBAR	Subtract A from a Register	00017-	
SBAS	Store Byte via A into System	000355	
SBRA	Subtract Register from A	00015-	
SBU	Scan Bytes Until	1266--	
SBW	Scan Bytes While	1264--	
SBX	Store Byte Extended	000407	
SBXX	Store Byte Extended, Indexed	0257--, 0267--	
SCMP	Set Code Map	000454	
SCPV	Set Current Process Variables	000463	*
SCS	Set Code Segment	000444	
SDA	Store Double via A	000363	
SDAS	Store Double via A into System	000353	
SDDX	Store Double Extended	000413	
SEND	Send	000065	*
SETE	Set ENV Register	000022	
SETL	Set L Register	000020	
SETP	Set P Register	000023	
SETS	Set S Register	000021	

Table A-1. Alphabetical List of Instructions (Continued)

Mnemonic	Description	Octal Code	
SFRZ	System Freeze	000053	*
SIOC	Store IOC	000460	*
SMAP	Set Map	000067	*
SMBP	Set Memory Breakpoint	000404	*
SNDQ	Signal a Send Is Queued	000052	*
SPT	Set Process Timer	000443	*
SQAS	Store Quadrupleword via A to SG	000446	*
SQX	Store Quadrupleword Extended	000415	
SRST	Soft Reset (NonStop TXP processor only)	000455	
SSW	Set Switches	000027	
STAR	Store A in Register	00011-	
STB	Store Byte	-54---	
STD	Store Double	-64---	
STOR	Store	-44---	
STRP	Set RP	00010-	
SVMP	Save Map Entries	000441	*
SWA	Store Word via A	000361	
SWAS	Store Word via A into System	000351	
SWX	Store Word Extended	000411	
SWXX	Store Word Extended, Indexed	0255--, 0265--	
SXBL	Set Extended Base and Limit	000427	*
TOTQ	Test OUTQ	000056	@
TPEF	Test Parity Error Freeze Circuits NonStop II processor only	000453	*
TRCE	Add Entry to Trace Table	000217	*
ULKX	Unlock Extended Memory	000431	*
UMPS	Unmap a Segment (NonStop II processor)	000043	*
VSEM	"V" a Semaphore	000077	*
VWCS	Verify LCS	000401	*
WSMT	Write to Operations and Service Processor	000437	*
WSPT	Write Segment Page Table Entry	000425	*
WWCS	Write to LCS	000400	*
XCAL	External Call	127---	
XCTR	XRAY Counter Bump	000033	*
XIOC	Exchange IOCs	000462	*
XMSK	Exchange Mask	000064	*
XOR	Exclusive OR	000012	
XSMG	Compute Checksum in Current Data	000343	
XSMX	Checksum Extended Block	000333	
ZERD	Zero Double	000002	

Table A-1. Alphabetical List of Instructions (Continued)

The one-character symbols immediately to the right of the instruction opcodes have the following meanings:

*	indicates a privileged instruction.
@	indicates an instruction designated for operating system use only.
$	indicates a decimal arithmetic optional instruction.
#	indicates a floating-point arithmetic optional instruction.

Table A-2. Categorized List of Instructions

16-Bit Arithmetic (Top of Register Stack)		
IADD	Integer Add	000210
LADD	Logical Add	000200
ISUB	Integer Subtract	000211
LSUB	Logical Subtract	000201
IMPY	Integer Multiply	000212
LMPY	Logical Multiply	000202
IDIV	Integer Divide	000213
LDIV	Logical Divide	000203
INEG	Integer Negate	000214
LNEG	Logical Negate	000204
ICMP	Integer Compare	000215
LCMP	Logical Compare	000205
CMPI	Integer Compare Immediate	001---
ADDI	Integer Add Immediate	104---
LADI	Logical Add Immediate	003---
32-Bit Signed Arithmetic		
CDI	Convert Double to Integer	000307
CID	Convert Integer to Double	000327
DADD	Double Add	000220
DSUB	Double Subtract	000221
DMPY	Double Multiply	000222
DDIV	Double Divide	000223
DNEG	Double Negate	000224

Table A-2. Categorized List of Instructions (Continued)

32-Bit Signed Arithmetic (continued)			
DCMP	Double Compare	000225	
DTST	Double Test	000031	
MOND	(Load) Minus One Double	000001	
ZERD	(Load) Zero Double	000002	
ONED	(Load) One Double	000003	
16-Bit Signed Arithmetic (Register Stack Element)			
ADRA	Add Register to A	00014-	
SBRA	Subtract Register from A	00015-	
ADAR	Add A to Register	00016-	
SBAR	Subtract A from Register	00017-	
ADXI	Add to Index Immediate	104---	
Decimal Arithmetic Load and Store			
QLD	Quadruple Load	00023-	
QST	Quadruple Store	00023-	
Decimal Integer Arithmetic			
QADD	Quadruple Add	000240	
QSUB	Quadruple Subtract	000241	
QMPY	Quadruple Multiply	000242	$
QDIV	Quadruple Divide	000243	$
QNEG	Quadruple Negate	000244	$
QCMP	Quadruple Compare	000245	$
Decimal Arithmetic Scaling and Rounding			
QUP	Quadruple Scale Up	00025-	
QDWN	Quadruple Scale Down	00025-	
QRND	Quadruple Round	000263	$
Decimal Arithmetic Conversions			
CQI	Convert Quad to Integer	000264	$
CQL	Convert Quad to Logical	000246	$
CQD	Convert Quad to Double	000247	$
CQA	Convert Quad to ASCII	000260	$
CIQ	Convert Integer to Quad	000266	$
CLQ	Convert Logical to Quad	000267	$
CDQ	Convert Double to Quad	000265	$
CAQ	Convert ASCII to Quad	000262	$
CAQV	Convert ASCII to Quad with Initial Value	000261	$
Floating-Point Arithmetic			
FADD	Floating-Point Add	000270	#
FSUB	Floating-Point Subtract	000271	#

Table A-2. Categorized List of Instructions (Continued)

FMPY	Floating-Point Multiply.....	000272	#
FDIV	Floating-Point Divide.....	000273	#
FNEG	Floating-Point Negate.....	000274	#
FCMP	Floating-Point Compare.....	000275	#
Extended Floating-Point Arithmetic			
EADD	Extended Floating-Point Add.....	000300	#
ESUB	Extended Floating-Point Subtract.....	000301	#
EMPY	Extended Floating-Point Multiply.....	000302	#
EDIV	Extended Floating-Point Divide.....	000303	#
ENEG	Extended Floating-Point Negate.....	000304	#
ECMP	Extended Floating-Point Compare.....	000305	#
Floating-Point Conversions			
CEF	Convert Extended to Floating.....	000276	#
CEFR	Convert Extended to Floating, Rounded...	000277	#
CFI	Convert Floating to Integer.....	000311	#
CFIR	Convert Floating to Integer, Rounded....	000310	#
CFD	Convert Floating to Double.....	000312	#
CFDR	Convert Floating to Double, Rounded.....	000313	#
CED	Convert Extended to Double.....	000314	#
CEDR	Convert Extended to Double, Rounded.....	000315	#
CEI	Convert Extended to Integer.....	000337	#
CEIR	Convert Extended to Integer, Rounded....	000316	#
CFQ	Convert Floating to Quad.....	000320	#
CFQR	Convert Floating to Quad, Rounded.....	000321	#
CEQ	Convert Extended to Quad.....	000322	#
CEQR	Convert Extended to Quad, Rounded.....	000323	#
CFE	Convert Floating to Extended.....	000325	#
CIF	Convert Integer to Floating.....	000331	#
CDF	Convert Double to Floating.....	000306	#
CDFR	Convert Double to Floating, Rounded.....	000326	#
CQF	Convert Quad to Floating.....	000324	#
CQFR	Convert Quad to Floating, Rounded.....	000330	#
CIE	Convert Integer to Extended.....	000332	#
CDE	Convert Double to Extended.....	000334	#
CQE	Convert Quad to Extended.....	000336	#
CQER	Convert Quad to Extended, Rounded.....	000335	#
Floating-Point Functionals			
IDX1	Calculate Index, 1 Dimension.....	000344	#
IDX2	Calculate Index, 2 Dimensions.....	000345	#
IDX3	Calculate Index, 3 Dimensions.....	000346	#
IDXP	Calculate Index, Bounds in Code Space...	000347	#
IDXD	Calculate Index, Bounds in Data Space...	000317	#

Table A-2. Categorized List of Instructions (Continued)

Register Stack Manipulation		
EXCH	Exchange A with B	000004
DXCH	Double Exchange	000005
DDUP	Double Duplicate	000006
STAR	Store A in a Register	00011-
NSAR	Nondestructive Store A in a Register	00012-
LDRA	Load A from a Register	00013-
LDI	Load Immediate	100---
LDXI	Load Index Immediate	10----
LDLI	Load Left Immediate	005---
Boolean Operations		
LAND	Logical AND	000010
LOR	Logical OR	000011
XOR	Exclusive OR	000012
NOT	NOT	000013
ORRI	OR Right Immediate	004---
ORLI	OR Left Immediate	0044--
ANRI	AND Right Immediate	006---
ANLI	AND Left Immediate	007---
Bit Shift and Deposit		
DPF	Deposit Field	000014
LLS	Logical Left Shift	0300--
DLLS	Double Logical Left Shift	1300--
LRS	Logical Right Shift	0301--
DLRS	Double Logical Right Shift	1301--
ALS	Arithmetic Left Shift	0302--
DALS	Double Arithmetic Left Shift	1302--
ARS	Arithmetic Right Shift	0303--
DARS	Double Arithmetic Right Shift	1303--
Byte Test		
BTST	Byte Test	000007
Memory Stack to/from Register Stack		
LWP	Load Word from Program	-2----
LBP	Load Byte from Program	-2-4--
PUSH	Push Registers to Memory	024---
POP	Pop Memory to Registers	124---
LWXX	Load Word Extended, Indexed	0254--, 0264--
SWXX	Store Word Extended, Indexed	0255--, 0265--

Table A-2. Categorized List of Instructions (Continued)

Memory Stack to/from Register Stack (continued)		
LBXX	Load Byte Extended, Indexed	0256--, 0266--
SBXX	Store Byte Extended, Indexed	0257--, 0267--
LDX	Load Index	-3----
NSTO	Nondestructive Store	-34---
LOAD	Load Word	-4----
STOR	Store Word	-44---
LDB	Load Byte	-5----
STB	Store Byte	-54---
LDD	Load Double	-6----
STD	Store Double	-64---
LADR	Load Address of Variable	-7----
ADM	Add to Memory	-74---
Load and Store via Address on Register Stack		
ANS	AND to SG Memory	000034
ORS	OR to SG Memory	000035
ANG	AND to Current Data	000044
ORG	OR to Current Data	000045
ANX	AND to Extended Memory	000046
ORX	OR to Extended Memory	000047
LWUC	Load Word from User Code Segment	000342
LWAS	Load Word via A from System	000350
LWA	Load Word via A	000360
SWAS	Store Word via A into System	000351
SWA	Store Word via A	000361
LDAS	Load Double via A from System	000352
LDA	Load Double via A	000362
SDAS	Store Double via A into System	000353
SDA	Store Double via A	000363
LBAS	Load Byte via A from System	000354
LBA	Load Byte via A	000364
SBAS	Store Byte via A into System	000355
SBA	Store Byte via A	000365
DFS	Deposit Field into System Data	000357
DFG	Deposit Field in Current Data	000367
LBX	Load Byte Extended	000406
SBX	Store Byte Extended	000407
LWX	Load Word Extended	000410
SWX	Store Word Extended	000411
LDDX	Load Doubleword Extended	000412
SDDX	Store Doubleword Extended	000413
LQX	Load Quadrupleword Extended	000414

Table A-2. Categorized List of Instructions (Continued)

Load and Store via Address on Register Stack (continued)			
	SQX	Store Quadrupleword Extended	000415	
	DFX	Deposit Field Extended	000416	
	SCS	Set Code Segment	000444	
	LQAS	Load Quadrupleword via A from SG	000445	*
	SQAS	Store Quadrupleword via A to SG	000446	*
Branching			
	BIC	Branch if Carry	-10---	
	BUN	Branch Unconditionally	-104--	
	BOX	Branch on Index	-1-4--	
	BGTR	Branch if CC Greater	-11---	
	BEQL	Branch if CC Equal	-12---	
	BGEQ	Branch if CC Greater or Equal	-13---	
	BLSS	Branch if CC Less	-14---	
	BAZ	Branch if A Zero	-144--	
	BNEQ	Branch if CC Not Equal	-15---	
	BANZ	Branch if A Not Zero	-154--	
	BLEQ	Branch if CC Less or Equal	-16---	
	BNOV	Branch if no Overflow	-164--	
	BNOC	Branch if no Carry	-17---	
	BFI	Branch Forward Indirect	000030	
Moves, Compares, Scans, and Checksum Computations			
	MNGG	Move Words While Not Duplicate	000226	
	CDG	Count Duplicate Words	000366	
	MOVW	Move Words	026---	
	MOVB	Move Bytes	126---	
	COMW	Compare Words	0262--	
	COMB	Compare Bytes	1262--	
	SBW	Scan Bytes While	1264--	
	SBU	Scan Bytes Until	1266--	
	MNDX	Move Words While Not Duplicate, Extended	000227	
	XSMX	Checksum Extended Block	000333	
	XSMG	Compute Checksum in Current Data	000343	
	CDX	Count Duplicate Words Extended	000356	
	MVBX	Move Bytes Extended	000417	
	MBXR	Move Bytes Extended Reverse	000420	
	MBXX	Move Bytes Extended, and Checksum	000421	
	CMBX	Compare Bytes Extended	000422	
Program Register Control			
	SETL	Set L Register	000020	
	SETS	Set S Register	000021	
	SETE	Set ENV Register	000022	

Table A-2. Categorized List of Instructions (Continued)

Program Register Control (continued)			
SETP	Set P Register	000023	
RDE	Read E Register	000024	
RDP	Read P Register	000025	
STRP	Set Register Pointer	00010-	
ADDS	Add to S Register	002---	
CCL	Set CC Less	000015	
CCE	Set CC Equal	000016	
CCG	Set CC Greater	000017	
Routine Calls/Returns			
PCAL	Procedure Call	027---	
XCAL	External Procedure Call	127---	
SCMP	Set Code Map	000454	
DPCL	Dynamic Procedure Call	000032	
EXIT	Exit from Procedure	125---	
DXIT	DEBUG Exit	000072	*
BSUB	Branch to Subprocedure	-174--	
RSUB	Return from Subprocedure	025---	
Interrupt System			
RIR	Reset INT Register	000063	*
XMSK	Exchange MASK Register	000064	*
IXIT	Exit from Interrupt Handler	000071	*
DISP	Dispatch	000073	*
RIBA	Read INTA and INTB Registers	000440	*
Bus Communication			
TOTQ	Test Out Queues	000056	@
SEND	Send Packet	000065	*
Input/Output			
RSW	Read Switch Register	000026	
SSW	Set Switch Register	000027	
EIO	Execute I/O	000060	*
IIO	Interrogate I/O	000061	*
HIIO	High-Priority Interrogate I/O	000062	*
RCHN	Reset I/O Channel	000447	*
LIOC	Load IOC	000457	*
SIOC	Store IOC	000460	*
XIOC	Exchange IOCs	000462	*
Miscellaneous			
NOP	No Operation	000000	
RCLK	Read Clock	000050	

Table A-2. Categorized List of Instructions (Continued)

Miscellaneous (continued)			
RCPU	Read Processor Number	000051	
BPT	Instruction Breakpoint Trap	000451	
RUS	Read Micro State	000461	*
BIKE	Bicycle While Idle	000464	*
Resource Management			
XCTR	XRAY Counter Bump	000033	*
MXON	Mutual Exclusion On	000040	*
MXFF	Mutual Exclusion Off	000041	*
SNDQ	Signal a Send Is Queued	000052	*
SFRZ	System Freeze	000053	*
DOFS	Disc Record Offset	000057	@
DLEN	Disc Record Length	000070	@
HALT	Processor Halt	000074	*
PSEM	"P" a Semaphore	000076	*
VSEM	"V" a Semaphore	000077	*
RPV	Read PROM Version Numbers (NonStop II)	000216	*
WWCS	Write LCS	000400	*
VWCS	Verify LCS	000401	*
RWCS	Read LCS	000402	*
FRST	Firmware Reset	000405	*
RSMT	Read from Operations & Service Processor	000436	*
WSMT	Write to Operations & Service Processor.	000437	*
RPT	Read Process Timer	000442	*
SPT	Set Process Timer	000443	*
BCLD	Bus Cold Load	000452	*
TPEF	Test Parity Error Freeze Circuits (NonStop II processor only)	000453	*
SRST	Soft Reset (NonStop TXP processor only).	000455	
DDTX	DDT Request (NonStop TXP processor only)	000456	*
Memory Management			
MAPS	Map In a Segment	000042	*
UMPS	Unmap a Segment (NonStop II processor)	000043	*
RMAP	Read Map (NonStop II processor only)	000066	*
SMAP	Set Map	000067	*
CRAX	Convert Relative to Absolute Extended	000423	*
RSPT	Read Segment Page Table Entry	000424	*
WSPT	Write Segment Page Table Entry	000425	*
RXBL	Read Extended Base and Limit	000426	*
SXBL	Set Extended Base and Limit	000427	*
LCKX	Lock Down Extended Memory	000430	*
ULKX	Unlock Extended Memory	000431	*
CMRW	Correctable Memory Error Read/Write	000432	*

Table A-2. Categorized List of Instructions (Continued)

Memory Management (continued)			
SVMP	Save Map Entries	000441	*
BNDW	Bounds Test Words	000450	*
SCPV	Set Current Process Variables	000463	*
ASPT	Address of Segment Page Table Header	000470	*
List Management			
DLTE	Delete Element from List	000054	*
INSR	Insert Element into List	000055	*
MRL	Merge onto Ready List	000075	*
FTL	Find Position in Time List	000206	*
DTL	Determine Time Left for Element	000207	*
Trace and Breakpoints			
TRCE	Add Entry to Trace Table	000217	*
SMBP	Set Memory Breakpoint	000404	*

The one-character symbols immediately to the right of the instruction opcodes have the following meanings:

- `*` indicates a privileged instruction.
- `@` indicates an instruction designated for operating system use only.
- `$` indicates a decimal arithmetic optional instruction.
- `#` indicates a floating-point arithmetic optional instruction.

Table A-3. Binary Coding, Memory Reference Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	vkcc
I	2	0 X X	0 +/- ←	P	→	LWP a
I	2	0 X X	1 +/- ←	P	→	LBP b
I	3	0 X X	←	G,L,SG,S	→	LDX a
I	3	1 X X	←	G,L,SG,S	→	NSTO
I	4	0 X X	←	G,L,SG,S	→	LOAD a
I	4	1 X X	←	G,L,SG,S	→	STOR
I	5	0 X X	←	G,L,SG,S	→	LDB b
I	5	1 X X	←	G,L,SG,S	→	STB
I	6	0 X X	←	G,L,SG,S	→	LDD a
I	6	1 X X	←	G,L,SG,S	→	STD
I	7	0 X X	←	G,L,SG,S	→	LADR
I	7	1 X X	←	G,L,SG,S	→	ADM vk a
		P+	0 .	. . .	. . .	0:177
		P-	1 .	. . .	. . .	0:177
		G+	0 . .	. . .	. . .	0:377
		L+	1 0 .	. . .	. . .	0:177
		SG	1 1 0	. . .	. . .	0:77
		L-	1 1 1	0 . .	. . .	0:37
		S-	1 1 1	1 . .	. . .	0:37

+/- (0/1) implies two's-complement notation; the sign is extended through bit 0 at execution.

I (0/1) indicates direct or indirect address.

v = Overflow

k = Carry

cc = Condition Codes:

a	L (result < 0) or (opr1 < opr2) E (result = 0) or (opr1 = opr2) G (result > 0) or (opr1 > opr2)	Note: opr1 is first item pushed on stack; opr2 is second.
b	L (ASCII numeric) E (ASCII alpha) G (ASCII special)	
c	L (channel error or timeout) E (no error) G (unusual condition)	

Table A-4. Binary Coding, Immediate Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	vkcc
1	0	0	+/- ←	OPERAND	→	LDI a
1	0	0 X X	+/- ←	OPERAND	→	LDXI a
0	0	1	+/- ←	OPERAND	→	CMPI a
0	0	2	+/- ←	OPERAND	→	ADDS a
0	0	3	+/- ←	OPERAND	→	LADI k a
0	0	4	0 ←	OPERAND	→	ORRI a
0	0	4	1 ←	OPERAND	→	ORLI a
1	0	4	+/- ←	OPERAND	→	ADDI vk a
1	0	1 X X	+/- ←	OPERAND	→	ADXI vk a
0	0	5	+/- ←	OPERAND	→	LDLI a
0	0	6	+/- ←	OPERAND	→	ANRI a
0	0	7	+/- ←	OPERAND	→	ANLI a

+/- (0/1) implies two's-complement notation; the sign is extended through bit 0 at execution.

I (0/1) indicates direct or indirect address.

vkcc: see Table A-3 footnote.

Table A-5. Binary Coding, Move/Shift/Call/Extended Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	vkcc
0	2	4	N	LAST	COUNT-1	PUSH
1	2	4	N	LAST	COUNT-1	POP
0	2	5	0 ←	SDEC	→	RSUB
1	2	5	0 ←	SDEC	→	EXIT
0	2	5/6	4	DISPLACEMENT		LWXX a
0	2	5/6	5	DISPLACEMENT		SWXX
0	2	5/6	6	DISPLACEMENT		LBXX b
0	2	5/6	7	DISPLACEMENT		SBXX
0	2	6	0 0 RL	S S D	RP	MOVW
0	2	6	0 1 RL	S S D	RP	COMW a
1	2	6	0 0 RL	S S D	RP	MOVB
1	2	6	0 1 RL	S S D	RP	COMB a
1	2	6	1 0 RL	S S D	RP	SBW k
1	2	6	1 1 RL	S S D	RP	SBU k
0	2	7	←	PEP	→	PCAL
1	2	7	←	PEP	→	XCAL
0	3	0	0	← SHIFT	COUNT →	LLS a
1	3	0	0	← SHIFT	COUNT →	DLLS a
0	3	0	1	← SHIFT	COUNT →	LRS a
1	3	0	1	← SHIFT	COUNT →	DLRS a
0	3	0	2	← SHIFT	COUNT →	ALS a
1	3	0	2	← SHIFT	COUNT →	DALS a
0	3	0	3	← SHIFT	COUNT →	ARS a
1	3	0	3	← SHIFT	COUNT →	DARS a

RL (right-left indicator)
0 left-to-right (increasing addresses)
1 right-to-left (decreasing addresses)

SS (source map):
00 Current Data
01 System Data (Current Data if nonprivileged user)
10 Current Code
11 User Code

D = (destination map), data only
0 Current Data
1 System Data (Current Data if Nonprivileged User)

PEP = Procedure Entry Point Table

SDEC = stack S decrement

vkcc: see Table A-3 footnote.

Table A-6. Binary Coding, Branch Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	vkcc
I	1	0	0 +/-	P	→	BIC
I	1	0	4 +/-	P	→	BUN
I	1	0 X X	4 +/-	P	→	BOX
I	1	1	0 +/-	P	→	BGTR
I	1	2	0 +/-	P	→	BEQL
I	1	3	0 +/-	P	→	BGEQ
I	1	4	0 +/-	P	→	BLSS
I	1	4	4 +/-	P	→	BAZ
I	1	5	0 +/-	P	→	BNEQ
I	1	5	4 +/-	P	→	BANZ
I	1	6	0 +/-	P	→	BLEQ
I	1	6	4 +/-	P	→	BNOV
I	1	7	0 +/-	P	→	BNOC
I	1	7	4 +/-	P	→	BSUB

+/- (0/1) implies two's-complement notation; the sign is extended through bit 0 at execution.

I (0/1) indicates direct or indirect address.

Note: since the Program Counter register holds the address of the next instruction, a branch-self instruction (Branch *) would be coded: BUN P-1.

vkcc: see Table A-3 footnote.

Table A-7. Binary Coding, Stack Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE		→

7:15>				vkcc
0	0	0	NOP	
0	0	1	MOND	a
0	0	2	ZERD	a
0	0	3	ONED	a
0	0	4	EXCH	a
0	0	5	DXCH	a
0	0	6	DDUP	a
0	0	7	BTST	b
0	1	0	LAND	a
0	1	1	LOR	a
0	1	2	XOR	a
0	1	3	NOT	a
0	1	4	DPF	a
0	1	5	CCL	a
0	1	6	CCE	a
0	1	7	CCG	a
0	2	0	SETL	
0	2	1	SETS	
0	2	2	SETE	!! !
0	2	3	SETP	
0	2	4	RDE	
0	2	5	RDP	
0	2	6	RSW	a
0	2	7	SSW	
0	3	0	BFI	
0	3	1	DTST	a
0	3	2	DPCL	
0	3	3	*XCTR	
0	3	4	ANS	a
0	3	5	ORS	a
0	4	0	*MXON	
0	4	1	*MXFF	
0	4	2	*MAPS	
0	4	3	*UMPS	
0	4	4	ANG	a
0	4	5	ORG	a
0	4	6	ANX	a
0	4	7	ORX	a
0	5	0	RCLK	

<7:15>				vkcc
0	5	1	*RCPU	
0	5	2	*SNDQ	
0	5	3	*SFRZ	
0	5	4	*DLTE	
0	5	5	*INSR	
0	5	6	@TOTQ	!
0	5	7	@DOFS	c
0	6	0	*EIO	c
0	6	1	*IIO	c
0	6	2	*HIIO	
0	6	3	*RIR	
0	6	4	*XMSK	
0	6	5	*SEND	!
0	6	6	*RMAP	
0	6	7	*SMAP	
0	7	0	@DLEN	
0	7	1	*IXIT	
0	7	2	*DXIT	
0	7	3	*DISP	
0	7	4	*HALT	
0	7	5	*MRL	
0	7	6	*PSEM	
0	7	7	*VSEM	
1	0	reg	STRP	
1	1	reg	STAR	
1	2	reg	NSAR	
1	3	reg	LDRA	a
1	4	reg	ADRA	vk a
1	5	reg	SBRA	vk
1	6	reg	ADAR	vk
1	7	reg	SBAR	vk a
2	0	0	LADD	k a
2	0	1	LSUB	k a
2	0	2	LMPY	v=0a
2	0	3	LDIV	v a
2	0	4	LNEG	k a
2	0	5	LCMP	a
2	0	6	*FTL	
2	0	7	*DTL	

Table A-7. Binary Coding, Stack Instructions (Continued)

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE →		

<7:15>				vkcc	<7:15>				vkcc
2	1	0	IADD	vk a	4	0	5	*FRST	
2	1	1	ISUB	vk a	4	0	6	LBX	b
2	1	2	IMPY	v a	4	0	7	SBX	
2	1	3	IDIV	v a	4	1	0	LWX	a
2	1	4	INEG	vk a	4	1	1	SWX	
2	1	5	ICMP	a	4	1	2	LDDX	a
2	1	6	*RPV		4	1	3	SDDX	
2	1	7	*TRCE		4	1	4	LQX	a
2	2	0	DADD	vk a	4	1	5	SQX	
2	2	1	DSUB	vk a	4	1	6	DFX	a
2	2	2	DMPY	vk a	4	1	7	MVBX	
2	2	3	DDIV	vk a	4	2	0	MBXR	
2	2	4	DNEG	vk a	4	2	1	MBXX	
2	2	5	DCMP	a	4	2	2	CMBX	!
2	2	6	MNGG	!	4	2	3	*CRAX	
2	2	7	MNDX	!	4	2	4	*RSPT	!
3	3	3	XSMX		4	2	5	*WSPT	
3	4	2	LWUC	a	4	2	6	*RXBL	
3	4	3	XSMG		4	2	7	*SXBL	
3	5	0	LWAS	a	4	3	0	*LCKX	!
3	5	1	SWAS		4	3	1	*ULKX	!
3	5	2	LDAS	a	4	3	2	*CMRW	!
3	5	3	SDAS		4	3	4	*RMEM	a
3	5	4	LBAS	b	4	3	5	*WMEM	
3	5	5	SBAS		4	3	6	*RSMT	
3	5	6	CDX		4	3	7	*WSMT	
3	5	7	DFS	a	4	4	0	*RIBA	
3	6	0	LWA	a	4	4	1	*SVMP	
3	6	1	SWA		4	4	4	SCS	
3	6	2	LDA	a	4	4	5	*LQAS	a
3	6	3	SDA		4	4	6	*SQAS	
3	6	4	LBA	b	4	4	7	*RCHN	!
3	6	5	SBA		4	5	0	*BNDW	!
3	6	6	CDG		4	5	1	BPT	
3	6	7	DFG	a	4	5	2	*BCLD	
4	0	0	*WWCS	!	4	5	3	*TPEF	
4	0	1	*VWCS	!	4	5	4	SCMP	
4	0	2	*RWCS		4	7	0	*ASPT	!
4	0	4	*SMBP						

Table A-7. Binary Coding, Stack Instructions (Continued)

```
* indicates a privileged instruction.
@ indicates an instruction designated for operating
    system use only.

vkcc: see Table A-3 footnote.

! = special vkcc meanings; see instruction definitions
    in Table B-1.
```

Table A-8. Binary Coding, Decimal Arithmetic Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE		→

<7:15>				vkcc	
2	3	0	+QST		
2	3	1	+QST x5		
2	3	2	+QST x6		
2	3	3	+QST x7		
2	3	4	+QLD		a
2	3	5	+QLD x5		a
2	3	6	+QLD x6		a
2	3	7	+QLD x7		a
2	4	0	+QADD	vk	a
2	4	1	+QSUB	vk	a
2	4	2	QMPY	v	a
2	4	3	QDIV	v	a
2	4	4	QNEG	vk	a
2	4	5	QCMP		a
2	4	6	CQL	v	
2	4	7	CQD	v	

<7:15>				vkcc	
2	5	0	+QUP	v	a
2	5	1	+QDWN	v=0	
2	5	2	+QUP(2)	v	a
2	5	3	+QDWN(2)	v=0	a
2	5	4	+QUP(3)	v	a
2	5	5	+QDWN(3)	v=0	a
2	5	6	+QUP(4)	v	a
2	5	7	+QDWN(4)	v=0	a
2	6	0	CQA	v	a
2	6	1	CAQV	v	!
2	6	2	CAQ	v	!
2	6	3	QRND	v=0	a
2	6	4	CQI	v	
2	6	5	CDQ		
2	6	6	CIQ		
2	6	7	CLQ		

```
+   indicates an instruction that is standard in all
      processors (not part of decimal option).

!   CCE if entire string is ASCII digits, CCG if not.

vkcc: see Table A-3 footnote.
```

Table A-9. Binary Coding, Floating-Point Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE		→

<7:15>				vkcc		<7:15>				vkcc
2	7	0	FADD	v	a	3	1	6	CEIR	a
2	7	1	FSUB	v	a	3	1	7	IDXD	a
2	7	2	FMPY	v	a	3	2	0	CFQ	a
2	7	3	FDIV	v	a	3	2	1	CFQR	a
2	7	4	FNEG		a	3	2	2	CEQ	a
2	7	5	FCMP		a	3	2	3	CEQR	a
2	7	6	CEF		a	3	2	4	CQF	a
2	7	7	CEFR		a	3	2	5	CFE	a
3	0	0	EADD	v	a	3	2	6	CDFR	a
3	0	1	ESUB	v	a	3	2	7	+CID	a
3	0	2	EMPY	v	a	3	3	0	CQFR	a
3	0	3	EDIV	v	a	3	3	1	CIF	a
3	0	4	ENEG		a	3	3	2	CIE	a
3	0	5	ECMP		a	3	3	4	CDE	a
3	0	6	CDF		a	3	3	5	CQER	a
3	0	7	+CDI		a	3	3	6	CQE	a
3	1	0	CFIR		a	3	3	7	CEI	a
3	1	1	CFI		a	3	4	4	IDX1	a
3	1	2	CFD		a	3	4	5	IDX2	a
3	1	3	CFDR		a	3	4	6	IDX3	a
3	1	4	CED		a	3	4	7	IDXP	a
3	1	5	CEDR		a					

\+ indicates an instruction that is standard in all processors (not part of floating-point option).

vkcc: see Table A-3 footnote.

APPENDIX B

INSTRUCTION SET DEFINITION

This appendix consists of two tables. Table B-1 is a key to the symbols used in the instruction definitions. Table (B-2) gives brief definitions of all the instructions in the NonStop II and NonStop TXP processors' instruction set, in numeric opcode order. A TAL-like notation is used for the definitions. This table is a specification of the instruction microcode, and is provided for those interested in microcode details such as the use of the Register Stack.

Table B-1. Definitions of Symbols

```
x&y=          bitwise "and" of x and y
x|y=          bitwise "or" of x and y
x xor y=      bitwise "exclusive or" of x and y
x mod y=      x modulo y
~ x=          bitwise "complement" of x
x<<n=         x arithmetically shifted left n bits
x>>n=         x arithmetically shifted right n bits
x'<<'n=       x logically shifted left n bits
x'>>'n=       x logically shifted right n bits
x rotate n=   x'<<'n + x.<0:n-1>
x:y=          if x<y then -1 else if x=y then 0 else 1
x'<'y=        comparison of x and y as 16-bit unsigned numbers
x':'y=        if x'<'y then -1 else if x=y then 0 else 1
x max y=      if x>y then x else y
x:=:y=        exchange x and y
x^y=          concatenate x and y

A=            R[RP]
address=      if indirect then mem[ memmap, dir.adr. ] else dir.adr.
              (*** NonStop II processor ***)
address=      if indirect then mem[ dseg, dir.adr. ] else dir.adr.
              (*** NonStop TXP processor ***)
```

Table B-1. Definitions of Symbols (Continued)

```
B=              R[RP-1]
BA.<0:31>=      B.<0:15>^A.<0:15>
binq[ bus,la ]= INQ[ bus, la.<0:14> ].byteflag
BKPT=           ENV.<1>
boq[ bus,la ]= OUTQ[ bus, la.<0:14> ].byteflag
                (*** NonStop II processor ***)
boq[ la ]=       OUTQ[ la.<0:14> ].byteflag
                (*** NonStop TXP processor ***)
BPADDR=         sysstack[ %115:%116 ]
BPADDRX=        sysstack[ %137 ]
BPBASE=         sysstack[ %123 ]
BPLIM=          sysstack[ %125 ]
BPSIZE=         sysstack[ %124 ]
branch=         P:=branch address
branch address= if indirect then code[dba] + dba else dba
BRT=            sysstack[ %1400:%1777 ]
bxmem[ xaddr ]= the byte at xaddr
byteaddress=    if indirect then mem[memmap,dir.adr.]+X
                else 2*dir.adr.+X
                (*** NonStop II processor ***)
byteaddress=    if indirect then mem[dseg,dir.adr.]+X
                else 2*dir.adr.+X
                (*** NonStop TXP processor ***)
bytedest[ la ]= mem[ destmap,la.<0:14> ].byteflag
                (*** NonStop II processor ***)
bytedest[ la ]= mem[ destseg,la.<0:14> ].byteflag
                (*** NonStop TXP processor ***)
byteflag=       <8*la.<15>:8*la.<15>+7>
bytesource[ la ]= mem[ srcmap, la.<0:14>+
                    (I.<10:11>=2)*P.<0>*%100000 ].byteflag
                    (*** NonStop II processor ***)
bytesource[ la ]= mem[ srcseg, la.<0:14>+
                    (I.<10:11>=2)*P.<0>*%100000 ].byteflag
                    (*** NonStop TXP processor ***)
bytex=          mem[ memmap, byteaddress.<0:14> ].byteflag
                (*** NonStop II processor ***)
bytex=          mem[ dseg, byteaddress.<0:14> ].byteflag
                (*** NonStop TXP processor ***)

C=              R[RP-2]
CACHE=          data/instruction cache
                (*** NonStop TXP processor ***)
CACHETAG=       tags for CACHE entries
                (*** NonStop TXP processor ***)
CB.<0:31>=      C.<0:15>^B.<0:15>
cc(x)=          Z:=(x=0); N:=(x<0)
ccb(x)=         Z:=("A"<=x<="Z") or ("a"<=x<="z"); N:=("0"<=x<="9")
CCE=            N:=0; Z:=1
CCG=            N:=0; Z:=0
CCL=            N:=1; Z:=0
ccl(x)=         cc(x); K:=adder carry
ccn(x)=         ccl(x); V:=adder overflow
ccz(x)=         Z:=(x=0); N:=0;
chkp(x)=        if  memory location "x" is absent  then  Page Fault
CLOCK=          sysstack[ %103:%106 ]
cmap=           LS*2+CS+2
                (*** NonStop II processor ***)
CMSEG=          (discontinued term; see CSSEG)
code[ la ]=     mem[ cmap, la ]
                (*** NonStop II processor ***)
code[ la ]=     mem[ cseg, la ]
                (*** NonStop TXP processor ***)
```

Table B-1. Definitions of Symbols (Continued)

```
computeshiftcount= if I.<10:15>=0 then {shiftcount:=A.<8:15>;
                     RP:=RP-1} else shiftcount:=I.<10:15>
CPCB=          sysstack[ %3 ]
CS=            ENV.<7>
cseg=          LS*2+CS+2
               (*** NonStop TXP processor ***)
CSPACEID       current space ID register
CSSEG=         sysstack[ %1340:%1357 ]
               a software copy of the SST register contents

D=             R[RP-3]
dba=           P+I.<9:15>-128*I.<8>
DC.<0:31>=     D.<0:15>^C.<0:15>
DCBA.<0:63>=   D.<0:15>^C.<0:15>^B.<0:15>^A.<0:15>
dest[ la ]=    mem[ destmap, la ]
               (*** NonStop II processor ***)
dest[ la ]=    mem[ destseg, la ]
               (*** NonStop TXP processor ***)
destmap=       if I.<12>&PRIV then 1 else DS
               (*** NonStop II processor ***)
destseg=       if I.<12>&PRIV then 1 else DS
               (*** NonStop TXP processor ***)
dir.adr.=      if I.<7>=0 then I.<8:15>      'global variable'
               else                              (0:255)
               if I.<8>=0 then L+I.<9:15>    'local variable'
               else                              (0:127)
               if I.<9>=0 then I.<10:15>     'system global'
               else                              (0:63)
               if I.<10>=0 then L-I.<11:15>  'procedure parameter'
               else                              (0:31)
                    S-I.<11:15>;             'subroutine parameter'
                                                 (0:31)
DS=            ENV.<6>
dseg=          if I.<7:9>=6 and PRIV then 1 else DS
               (*** NonStop TXP processor ***)
dwordx=        mem[ memmap, address+2*X:address+2*X+1 ]
               (*** NonStop II processor ***)
dwordx=        mem[ dseg, address+2*X:address+2*X+1 ]
               (*** NonStop II processor ***)

E=             R[RP-4]
ECS=           entry control store, first vertical control store
               word for each instruction
               (*** NonStop TXP processor ***)
ED.<0:31>=     E.<0:15>^D.<0:15>
ENV.<0:15>=    environment register
EPT=           entry point table for instruction decoding
extended address=  segment ^ page ^ word ^ byte

F=             R[RP-5]
FE.<0:31>=     F.<0:15>^E.<0:15>

G=             R[RP-6]

H=             R[RP-7]
HCS=           horizontal control store
               (*** NonStop TXP processor ***)
HGFE.<0:63>=   H.<0:15>^G.<0:15>^F.<0:15>^E.<0:15>
hit(xa)=       if block of memory starting at "xa" is in CACHE
               (and valid) then true else false
               (*** NonStop TXP processor ***)
```

Table B-1. Definitions of Symbols (Continued)

```
I.<0:15>=     instruction register
imm=          I.<8:15>-256*I.<7>
indirect=     I.<0>
INQ[0:1,0:15].<0:15>= interprocessor bus in queues
INTA.<0:15>= interrupt register A
INTB.<0:15>= interrupt register B
IOC=          sysstack[ %2000:%3777 ]
IOCSPAD=      IOC scratchpad registers (IOC cache)
              (*** NonStop TXP processor ***)

K=            ENV.<9>

L.<0:15>=     local data pointer=location of current stack marker
LIGHTS.<0:15>= switch register output
LS=           ENV.<4>

MAP[0:15,0:63].<0:15>= memory map
                       (*** NonStop II processor ***)
MASK.<0:15>= interrupt mask register
mem[ m,a ]=  MEMORY[ MAP[ m,a.<0:5> ].<0:12>, a.<6:15> ]
              (*** NonStop II processor ***)
mem[ sas,la ]=  xmem[ axaddr( SST[ sas ], la, 0 ) ]
                 (*** NonStop TXP processor ***)
memmap=       if I.<7:9>=6 and PRIV then 1 else DS
              (*** NonStop II processor ***)
MEMORY[0:8191,0:1023].<0:15>= physical memory
movestep=     if I.<9> then -1 else 1
MYEXTCPU=     sysstack[ %154 ]
                .<8:11>=  cluster number
                .<12:15>= processor number

N=            ENV.<11>

OUTQ[0:1,0:15].<0:15>= interprocessor bus out queues
                       (*** NonStop II processor ***)
OUTQ[0:15].<0:15>=     interprocessor bus out queue
                       (*** NonStop II processor ***)
P.<0:15>=     program counter=1+location of current instruction
PCACHE=       page table cache
              (*** NonStop TXP processor ***)
PCACHETAG=    tags for PCACHE entries
              (*** NonStop TXP processor ***)
PHYPAGE=      mem[ %16, %150000:%167777 ]
PHYSEG=       mem[ %16, %130000:%147777 ]
PRIV=         ENV.<5>
PRIV TRAP=    cause an instruction failure interrupt
ptchit(xa)=   if page table entry for "xa" is in PCACHE
              (and valid) then true else false
              (*** NonStop TXP processor ***)
ptfill(xa)=   {x := mem[ SEG[ xa.<2:14>*2 ].<5:8>,
                         SEG[ xa.<2:14>*2+1 ]+xa.<15:20> ];
              if ~x.<15> then   ! entry is valid, set "Referenced"
                    mem[ SEG[ xa.<2:14>*2 ].<5:8>,
                         SEG[ xa.<2:14>*2+1 ]+xa.<15:20> ]:=
                    x := x | %4;
              PCACHE[ xa.<2:14>, xa.<15:20> ] := x;
               PCACHETAG[ xa.<2:14>, xa.<15:20> ] :=
                (x & %174003) | (xa.<2:10> << 2) }}
              (*** NonStop TXP processor ***)
PTIME=        sysstack[%126:%127]
```

Table B-1. Definitions of Symbols (Continued)

```
ptmiss(xa)=  ~ptchit(xa)
             (*** NonStop TXP processor ***)

RLIST=       sysstack[ %100:%101 ]
roma=        program counter for instruction microprocessor
             (*** NonStop II processor ***)
RP=          ENV.<13:15>

S.<0:15>=    stack pointer=location of last word of stack
sas=         short address space (range 0-15)
SD=          scratch register.  When the processor is in the idle
             loop, it will indicate the reason:
                  %000000     interrupt occurred before LCS loaded
                  %000001     LCS opcode used before LCS loaded
                  %000003     tape dump attempted
                  %000014     bus cold load sequence error
                  %000040     manual reset
                  %000053     SFRZ instruction
                  %000074     HALT instruction
                  %000100     DDT halt interrupt
                  %000115     OSP memory access breakpoint
                  %000200     halt interrupt
                  %000377     bus cold load checksum error
                  %001000     i/o channel timeout on a cold load
                  %001154     memory dump completed
                  %002000     power-on interrupt with invalid memory
                  %100000     an UCME occurred when masked off
                  %100001     a DABS occurred when masked off
                  %100002     an IABS occurred when masked off
                  %100003     a microcode or hardware failure occurred
                  %100004     an error (CCG or CCL) occurred during
                              the coldload EIO
                  %100010     an instruction failure occurred before
                              LCS was loaded
                  %100011     a stack overflow occurred before
                              LCS was loaded
                  %100012     Hardware failure <type>
                  %100013     Hardware failure <type>
                  %100014     Hardware failure <type>
                  %100015     Hardware failure: IPU parity checker
                  %100016     Hardware failure: MCB parity checker
                  %100017     Hardware failure: CCD parity checker
                  %100020     Hardware failure: suspect IPU board
                  %177771     Model 3206 tape controller firmware
                              not loaded
                  %177772     illegal cold load switch setting
                  %177773     i/o channel timeout on a tape dump
                  %177774     error during memory dump to tape
                  %177775     interrupt during memory dump to
                              interprocessor bus
                  %177776     uncorrectable memory error during map
                              recovery following a power-on
                  %177777     spurious interrupt
SEG=          mem[ 14, %70000:%127777 ]
segment base= MAP[ 14, 60:61 ]
              (*** NonStop II processor ***)
segment limit= MAP[ 14, 62:63 ]
              (*** NonStop II processor ***)
SEGTABSIZE=   sysstack[ %65 ]
SIV=          sysstack[ %1200:%1337 ]
source[ la ]= mem[ srcmap, la ]
              (*** NonStop II processor ***)
```

Table B-1. Definitions of Symbols (Continued)

```
source[ la ]= mem[ srcseg, la ]
              (*** NonStop TXP processor ***)
srcmap=       if I.<10> then {if I.<11> then 2 else cmap}
              else if I.<11>&PRIV then 1 else DS
              (*** NonStop II processor ***)
srcseg=       case I.<10:11> of
                begin
                 DS;                              ! current data
                 if PRIV then 1 else DS;          ! system data if PRIV
                 cseg;                            ! current code
                 2;                               ! user code
                end;
              (*** NonStop TXP processor ***)
SST=          hardware Short Segment Table registers
              (*** NonStop TXP processor ***)
stack[ la ]=     mem[ DS, la ]
SWITCHES.<0:15>= switch register input
sysstack[ la ]=  mem[ 1, la ]

T=            ENV.<8>
TLIST=        sysstack[ %107:%110 ]
TRACE=        sysstack[ %121 ]
TRBASE=       sysstack[ %117 ]
TRLIM=        sysstack[ %120 ]

UC=            ENV.<0>
UCOPTIONFLAG= sysstack[ %130 ]
UC^BASE=      user code segment base register
UC^SIZE=      user code space size register
UL^BASE=      user library segment base register
UL^SIZE=      user library space size register

V=            ENV.<10>
VCS=          vertical control store
              (*** NonStop TXP processor ***)

WCS=          writable control store
word=         mem[ memmap, address ]
              (*** NonStop II processor ***)
word=         mem[ dseg, address ]
              (*** NonStop TXP processor ***)
wordx=        mem[ memmap, address+X ]
              (*** NonStop II processor ***)
wordx=        mem[ dseg, address+X ]
              (*** NonStop TXP processor ***)

X=            if I.<5:6>=0 then 0 else R[I.<5:6>+4]
xaddr.<0:31>= a 32-bit extended address
XB=           extended address base address register
xbase=        stack[ L*I.<5>+I.<10:15> : L*I.<5>+I.<10:15>+1 ]
XL=           extended address limit register
```

Table B-1. Definitions of Symbols (Continued)

```
xmap=          cross code space map
                !parameter=new space ID
               CSPACEID:=parameter;
               m:=CSPACEID.<4>*2+CSPACEID.<7>+2;
               case m-2 of
                {
               !0!usercode!
                 {if CSPACEID.<11:15> >=UC^SIZE then
                      {instruction failure};
                  seg:=UC^BASE+CSPACEID.<11:15>
                 };
               !1!system code!
                 {if CSPACEID.<11:15> <>0 then
                      {instruction failure};
                  seg:=3;
                 };
               !2!user library!
                 {if CSPACEID.<11:15> >=UL^SIZE then
                      {instruction failure};
                  seg:=UL^BASE+CSPACEID.<11:15>
                 };
               !3!system library!
                 %34+CSPACEID.<11:15>
                };
               if CSSEG[m]<>seg then
                call MAPS(seg,m).
xmem[ xaddr ]= the word located at xaddr

Z=             ENV.<12>
```

Table B-2. Instruction Definitions

Note: The one-character symbols immediately to the right of the instruction opcodes have the following meanings:

- `*` indicates a privileged instruction.
- `@` indicates an instruction designated for operating system use only.
- `$` indicates a decimal arithmetic optional instruction.
- `#` indicates a floating-point arithmetic optional instruction.
- `%` indicates an instruction for NonStop II processors only.
- `&` indicates an instruction for NonStop TXP processors only.

op(x) indicates that an operation similar to that performed by the instruction 'op' should be done using the value(s) 'x'.

0	0	0	0	0	0	NOP	no operation	
0	0	0	0	0	1	MOND	minus one double	RP:=RP+2; cc(B:=A:=-1)
0	0	0	0	0	2	ZERD	zero double	RP:=RP+2; cc(B:=A:=0)
0	0	0	0	0	3	ONED	one double	RP:=RP+2; B:=0; cc(A:=1)
0	0	0	0	0	4	EXCH	exchange	A:=:B; cc(A)
0	0	0	0	0	5	DXCH	double exchange	BA:=:CD; cc(BA)
0	0	0	0	0	6	DDUP	double duplicate	RP:=RP+2; cc(BA:=DC)
0	0	0	0	0	7	BTST	byte test	ccb(A.<8:15>); RP:=RP-1
0	0	0	0	1	0	LAND	logical AND	cc(B:=B&A); RP:=RP-1
0	0	0	0	1	1	LOR	logical OR	cc(B:=B|A); RP:=RP-1
0	0	0	0	1	2	XOR	exclusive OR	cc(B:=B xor A); RP:=RP-1
0	0	0	0	1	3	NOT	logical NOT	cc(A:= ~ A)
0	0	0	0	1	4	DPF	deposit field	cc(C:=(C&B | A&~B)); RP:=RP-2
0	0	0	0	1	5	CCL	cond. code less	Z:=0; N:=1
0	0	0	0	1	6	CCE	cond. code equal	Z:=1; N:=0
0	0	0	0	1	7	CCG	cond. code greater	Z:=N:=0
0	0	0	0	2	0	SETL	set L register	L:=A; RP:=RP-1
0	0	0	0	2	1	SETS	set S register	S:=A; RP:=RP-1
0	0	0	0	2	2	SETE	set ENV register	ENV.<0:7>:=ENV.<0:7>&A.<0:7>; ENV.<8:15>:=A.<8:15>
0	0	0	0	2	3	SETP	set P register	P:=A; RP:=RP-1
0	0	0	0	2	4	RDE	read ENV register	RP:=RP+1; A:=ENV
0	0	0	0	2	5	RDP	read P register	RP:=RP+1; A:=P
0	0	0	0	2	6	RSW	read switches	RP:=RP+1; cc(A:=SWITCHES)
0	0	0	0	2	7	SSW	set switches	sysstack[%122]:=LIGHTS:=A; RP:=RP-1
0	0	0	0	3	0	BFI	branch forward indirect	P:=P+A+code[P+A]; RP:=RP-1
0	0	0	0	3	1	DTST	double test	cc(BA)

0	0	0	0	3	2	DPCL	dynamic procedure call
							
							A.<0:6>=spaceid
							A.<7:15>=pep index
								t:=(ENV&%177740)
								stack[S+1:S+3]:=(P,t,L);
								t.<7>:=A.<0>;
								t.<4>:=A.<1>;
								t.<11:15>:=A.<2:6>;
								
								call xmap(t);
								m: A.<0>+2*A.<1>+2;
								t: A.<7:15>;
									CSPACEID;
									
									!CS
									!LS
									!space
									! index

Table B-2. Instruction Definitions (Continued)

				if ~PRIV then
				{if t>=mem[m,0] then
				{if t>=mem[m,1] then
				priv trap;
				PRIV:1;
				}
				};
				L:=S:=S+3;
				CS:=A.<0>;
				LS:=A.<1>;
				P:=code[t];
				RP:=7.

0 0	0	0	3 3*	XCTR	XRAY counter bump
					E=parameter
					(if A.<10:13>=1)
					DC=ext addr
					(if A.<4:6>=3)
					C=cntr blk addr
					(if A.<4:6> <> 3)
					B=cntr offset
					A=action
					<4:6> addr mode
					<10:13> action
						if (t:=(if A.<4:6>=0 then
						sysstack[B]
						else
						if A.<4:6>=1 then stack[B]
						else
						if A.<4:6>=2 then
						sysstack[B]
						else
						if A.<4:6>=3 then
						xmem[CB])) <> 0 then
						{if A.<4:6> = 0 then
						! system data space
						a:=%40001^(t+D)^0
						else
						! absolute segment zero
						a:=%40000^(t+D)^0;
						if A.<10:13>=0 then
						! increment
						{xmem[a:a+3]:=xmem[a:a+3]+1}
						else if A.<10:13>=1 then
						! add parameter
						{xmem[a:a+3]:=xmem[a:a+3]
						+E;
						if E<0 and xmem[a:a+1]<0
						then xmem[a:a+3]:=0}
						else
						{clock:=sysstack[%103:%106]
						+microsecond counter;
						if A.<10:13>=2 then
						! set state
						{if xmem[a:a+1]=0 then
						{xmem[a:a+1]:=1;
						a:=a+2;
						xmem[a:a+7]:=xmem[a:a+7]
						-clock}}
						else if A.<10:13>=3 then
						! reset state
						{if xmem[a:a+1]=1 then
						{xmem[a:a+1]:=0;
						a:=a+2;
						xmem[a:a+7]:=xmem[a:a+7]
						+clock}}

Table B-2. Instruction Definitions (Continued)

```
                                |     |                    |    else if A.<10:13>=4 then
                                |     |                    |     ! increment state
                                |     |                    |    {if xmem[a:a+1]<16384 then
                                |     |                    |      {t:=xmem[a:a+1]:=
                                |     |                    |          xmem[a:a+1]+1;
                                |     |                    |      a:=a+2;
                                |     |                    |      if xmem[a-2:a-1]<t then
                                |     |                    |       xmem[a-2:a-1] := t;
                                |     |                    |      a:=a+2;
                                |     |                    |      xmem[a:a+7]:=xmem[a:a+7]
                                |     |                    |       -clock}}
                                |     |                    |    else if A.<10:13>=5 then
                                |     |                    |     ! decrement state
                                |     |                    |     {if xmem[a:a+1]>0 then
                                |     |                    |      {xmem[a:a+1]
                                |     |                    |        := xmem[a:a+1]-1;
                                |     |                    |      a:=a+4;
                                |     |                    |      xmem[a:a+7]:=xmem[a:a+7]
                                |     |                    |       +clock}}
                                |     |                    |    }};
                                |     |                    |RP:=RP-3;
                                |     |                    |if A.<4:6>=3 then RP:=RP-1;
                                |     |                    |if A.<10:13>=1 then RP:=RP-1;
                                |     |                    |** NOTE: All counters must be
                                |     |                    |   present.
                                |     |                    |** NOTE: Counters may not
                                |     |                    |   cross page boundaries.
0 0  0  0  3  4 |ANS  |AND to SG memory    |cc(dest(A):=dest(A) & B);
                |     |                    |RP:=RP-2
0 0  0  0  3  5 |ORS  |OR to SG memory     |cc(dest(A):=dest(A) | B);
                |     |                    |RP:=RP-2
0 0  0  0  3  6 |     |                    | *** undefined ***
0 0  0  0  3  7 |     |                    | *** undefined ***
0 0  0  0  4  0*|MXON |mutual exclusion    |chkp(stack[(L-20) max 0]);
                |     |on                  |chkp(stack[S+A.<8:15>]);
                |     |A=<0:7> code size   |if A.<0:7>
                |     |  <8:15>stack size  |then chkp(code[P+A.<0:7>]);
                |     |                    |stack[L+1]:=MASK;
                |     |                    |MASK:=MASK & %177640;
                |     |                    |RP:=RP-1
0 0  0  0  4  1*|MXFF |mutual exclusion    |MASK:=stack[L+1]
                |     |off                 |
0 0  0  0  4  2*|MAPS |map in a segment    |! Only unmap data segments
                %|     |                    |if CMSEG[A]<>B and CMSEG[A]
                |     |A=map number        |                    <>-1 then
                |     |B=segment number    | {if A<=1 or A>=6 then
                |     |                    |   UMPS(A)
                |     |                    |  else
                |     |                    |   SEG[CMSEG[A]*2].<0:4>:=-1
                |     |                    |  };
                |     |                    |j:=B*2;
                |     |                    |i:=0;
                |     |                    |if B<>-1 then
                |     |                    | {if SEG[j].<0:4> <=15 then
                |     |                    |     instruction failure;
                |     |                    |  if A<=1 or A>=6 then
                |     |                    |   for i:=32 to $min(64,32+
                |     |                    |            SEG[j].<9:15>) do
                |     |                    |    {if MAP[15,i].<0:14>=b
                |     |                    |                      then
```

Table B-2. Instruction Definitions (Continued)

								{t:=MAP[15,i-32];
								mem[SEG[j].<5:8>,
								SEG[j+1]+i-32+
								MAP[15,i].<15>*32]
								:=t;
								MAP[15,i]:=-1;
								}
								};
								while i<SEG[j].<9:15> do
								{MAP[A,i]: mem[SEG[j]
								.<5:8>,SEG[j+1]+i];
								i:=i+1
								};
								SEG[j].<0:4>:=A;
								};

								while i<=63 do {MAP[A,i]:=1; i:=i+1}; CMSEG[A]:=B; RP:=RP-2. !Note!the page table must be !in memory
0	0	0	0	4	2* &	MAPS	"map" a segment into SST B=segment number A=SST entry #	SST[A]:=CSSEG[A]:=B RP:=RP-2

0	0	0	0	4	3*
					%
						UMPS	unmap a segment
							A=map number
								j:= SEG[CMSEG[A]*2].<9:15>;
								m:= SEG[CMSEG[A]*2].<5:8>
								p:= SEG[CMSEG[A]*2+1];
								for i := 0 to j-1 do
								{mem[m,p+i]:=t:=MAP[A,i];
								SEG[CMSEG[A]*2].<0:4>:=%37;
								CMSEG[A] := -1;
								RP:=RP-1
								!Note!the page table must be
								!in memory

0	0	0	0	4	4	ANG	AND to memory	cc(stack[A]:=stack[A] & B); RP:=RP-2
0	0	0	0	4	5	ORG	OR to memory	cc(stack[A]:=stack[A] | B); RP:=RP-2
0	0	0	0	4	6	ANX	AND to extended memory	cc(xmem[BA]:=xmem[BA] & C); RP:=RP-3
0	0	0	0	4	7	ORX	OR to extended memory	cc(xmem[BA]:=xmem[BA] | C); RP:=RP-3
0	0	0	0	5	0	RCLK	read clock	RP:=RP+4; DCBA:=sysstack[%103:%106]+ microsecond counter
0	0	0	0	5	1	RCPU	read processor #	RP:=RP+1; A:=processor #
0	0	0	0	5	2*	SNDQ	signal that a SEND is queued	set dispatcher interrupt; sysstack[%1277].<14>:=1
0	0	0	0	5	3*	SFRZ	system freeze	assert system freeze; halt

Table B-2. Instruction Definitions (Continued)

0	0	0	0	5	4*	DLTE	delete an element from a doubly linked, circular list
							
							A=element address
								if sysstack[A] <> 0 then
								{if sysstack[sysstack[A]+1]
								<> A or
								sysstack[sysstack[A+1]]
								<> A
								then Instruction Failure;
								f:=sysstack[A];
								b:=sysstack[A+1];
								sysstack[b]:=f;
								sysstack[f+1]:=b;
								sysstack[A]:=0;
								sysstack[A+1]:=0;
								};
								RP:=RP-1
								!!! Note !!!
								all memory locations accessed must be present

0	0	0	0	5	5*	INSR	insert an element into a doubly linked, circular list
							
							B=list header
							A=list element
								if A=0 or
								sysstack[sysstack[B]+1]
								<> B or
								sysstack[sysstack[B+1]]
								<> B
								then Instruction Failure;
								f:=sysstack[B];
								sysstack[B]:=A;
								sysstack[A]:=f;
								sysstack[A+1]:=B;
								sysstack[f+1]:=A;
								RP:=RP-2
								!!! Note !!!
								all memory locations accessed must be present

0	0	0	0	5	6@%	TOTQ	test out queues	N:=0; Z:=1; if either OUTQ full then Z:=0
0	0	0	0	5	6@&	TOTQ	test OUTQ	N:=0; Z:=1; if ~OUTQ empty then Z:=0
0	0	0	0	5	7@	DOFS	disc record offset A=record number on return, A holds offset into buffer of record	if A'>='512 or (A:=xmem[stack[L+2:3]-A*2]) '>='stack[L+4] then {P:=stack[L+5]; RP:=7};

0	0	0	0	6	0*	EIO	execute i/o	ioselect(A.subchannel);
								iocontrol(A.command,B);
								B:='device status';
								cc(A:='channel status')

0	0	0	0	6	1*	IIO	interrogate i/o	RP:=RP+3; C:='interrupt cause'; B:='interrupt status'; cc(A:='channel status');
0	0	0	0	6	2*	HIIO	high-priority interrogate i/o	RP:=RP+3; C:='high-priority interrupt cause'; B:='high-priority interrupt status'; cc(A:='channel status');
0	0	0	0	6	3*	RIR	reset interrupt register	'clear interrupt' A.<12:15> RP:=RP-1
0	0	0	0	6	4*	XMSK	exchange mask	MASK:=:A

Table B-2. Instruction Definitions (Continued)

```
0 0  0  0  6  5*|SEND|send                |do
                %|    |                   |{do until OUTQEMPTY or
                 |    |G=<15> bus         |        .8(32768-D) microsec;
                 |    |F=sequence #       | if OUTQEMPTY then
                 |    |E=<0:7> sender     | {if A<>0 then
                 |    |         cpu #     |  {bus:=G.<15>
                 |    |  <8:15> receiver  |   receiver:=E.<8:15>;
                 |    |          cpu #    |   OUTQ[bus,0]:=E;
                 |    |D=OUTQ full timer  |   OUTQ[bus,1]:=F;
                 |    |CB=buffer address  |   for i:=4 to 29 do
                 |    |A=byte count       |    {if A <> 0  then
                 |    |                   |     {boq[bus,i]:=bxmem[CB];
                 |    |                   |      A:=A-1; CB:=CB+1}
                 |    |                   |     else boq[bus,i]:=0};
                 |    |                   |   OUTQ[bus,15]:=(-1) xor
                 |    |                   |    OUTQ[bus,0]
                 |    |                   |             ... OUTQ[bus,14];
                 |    |                   |   if E.<8:11> then
                 |    |                   |    OUTQ[bus,15]:=
                 |    |                   |     OUTQ[bus,15] xor
                 |    |                   |     (E&%170000) xor
                 |    |                   |     (MYEXTCPU.<8:11>'<<'8);
                 |    |                   |   D:=0;
                 |    |                   |   if (F:=F+1)=0 then
                 |    |                   |    {done:=true; N:=0; Z:=1};
                 |    |                   |  } else
                 |    |                   |  {done:=true; N:=0; Z:=1
                 |    |                   |  }
                 |    |                   | } else
                 |    |                   | {done:=true; N:=1; Z:=0;
                 |    |                   |  OUTQEMPTY:=true
                 |    |                   | };
                 |    |                   |} until done;
                 |    |                   |RP:=RP-7
                 |    |                   |!!! Note !!!
                 |    |                   |xmem[CB:CB+A*2-1] must be
                 |    |                   |in memory
0 0  0  0  6  5*|SEND|send                |do
                &|    |                   |{do until OUTQEMPTY or
                 |    |G=<15> bus         |      .833(32768-D) microsec;
                 |    |F=sequence #       | if OUTQEMPTY then
                 |    |E=<0:7> sender     | {if A<>0 then
                 |    |         cpu #     |  {bus:=G.<15>
                 |    |  <8:15> receiver  |   receiver:=E.<8:15>;
                 |    |          cpu #    |   OUTQ[0]:=E;
                 |    |D=OUTQ full timer  |   OUTQ[1]:=F;
                 |    |CB=buffer address  |   for i:=4 to 29 do
                 |    |A=byte count       |    {if A <> 0  then
                 |    |                   |     {boq[i]:=bxmem[CB];
                 |    |                   |      A:=A-1; CB:=CB+1}
                 |    |                   |     else boq[i]:=0};
                 |    |                   |   OUTQ[15]:=(-1) xor
                 |    |                   |    OUTQ[0] xor OUTQ[1] ...
                 |    |                   |    ... xor OUTQ[14];
                 |    |                   |   if E.<8:11> then
                 |    |                   |    OUTQ[15]:=OUTQ[15] xor
                 |    |                   |     (E&%170000) xor
                 |    |                   |     (MYEXTCPU.<8:11>'<<'8);
                 |    |                   |   D:=0;
```

Table B-2. Instruction Definitions (Continued)

								if (F:=F+1)=0 then
								{done:=true; N:=0; Z:=1};
								} else
								{done:=true; N:=0; Z:=1
								}
								} else
								{done:=true; N:=1; Z:=0;
								OUTQEMPTY:=true
								};
								} until done;
								RP:=RP-7
								!!! Note !!!
								xmem[CB:CB+A*2-1] must be
								in memory

0	0	0	0	6	6* %	RMAP	read map	A:=MAP[A.<12:15>,A.<0:5>]
0	0	0	0	6	7* %	SMAP	set map	MAP[A.<12:15>,A.<0:5>]:=B; RP:=RP-2

0	0	0	0	6	7*
					&
						SMAP	set map
							B=entry
							A.<0:5>=logical page
							A.<12:15>=SST index
								s:=SST[A.<12:15>];
								p:=A.<0:5>;
								PCACHE[s,p]:=B;
								PCACHETAG[s,p]:=
								(PCACHE[s,p]&174003)
								|(s.<3:11><<2);
								xa:=0D; xa.<0>:=1;
								xa.<1:14>:=s;
								xa.<15:20>:=p;
								for i:=0 to 127 do
								{if hit(xa) then
								invalidate entry;
								xa:=xa+%20};
								RP:=RP-2
								!!! Note !!!
								WSPT must be used once the
								page tables are set up

0	0	0	0	7	0@	DLEN	disc record length A=record number	if (A:=DOFS(A+1)-DOFS(A)) < 0 then {P:=stack[L+5]; RP:=7}

0	0	0	0	7	1*	IXIT	interrupt exit	CSPACEID:=sysstack[L-5]&
								%4437;
								(MASK,S,P,ENV,L):=
								sysstack[L-4:L];
								call xmap(CSPACEID);
								R[0:7]:=sysstack[L+1:L+8];
								if not DS then
								{PTIME:=PTIME-TIMER-(10000*
								INTA.<13>)}.
								!Note!sysstack[L-5:L+8] must
								!be present

0	0	0	0	7	2*	DXIT	DEBUG exit	S:=L-6;
								(P,ENV,L): sysstack[L-2:L];
								CSPACEID:=sysstack[L-5]&
								%4437;
								call xmap(CSPACEID);
								if ENV.<0> then
								instruction breakpoint.

Table B-2. Instruction Definitions (Continued)

```
0 0  0  0  7  3*|DISP|dispatch          |set dispatcher interrupt;
                |    |                  |sysstack[%1277].<15>:=1
0 0  0  0  7  4*|HALT|processor halt     |halt
0 0  0  0  7  5*|MRL |merge onto ready    |t := sysstack[ %101 ];
                |    |list               |while sysstack[t+2].<8:15> <
                |    |A=PCB address      |      sysstack[A+2].<8:15>
                |    |                   |do t:=sysstack[t+1];
                |    |                   |if sysstack[CPCB+2].<8:15> <
                |    |                   |   sysstack[A+2].<8:15>
                |    |                   |then DISP;
                |    |                   |insert A after t; RP:=RP-1
0 0  0  0  7  6*|PSEM|"P" a semaphore     |sysstack[A+2]:=sysstack[A+2]
                |    |                   |                -1;
                |    |CB=wait time       |if < then
                |    |A=semaphore addr   |  {set dispatcher interrupt;
                |    |                   |   sysstack[%1277]:=
                |    |                   |         sysstack[%1277] | 5}
                |    |                   |else {C:=1;
                |    |                   |      sysstack[A+3]:=CPCB};
                |    |                   |RP:=RP-2
                |    |                   | !!! Note !!!
                |    |                   | sysstack must be resident
0 0  0  0  7  7*|VSEM|"V" a semaphore     |sysstack[A+2]:=sysstack[A+2]
                |    |                   |                +1;
                |    |A=semaphore addr   |if <= then
                |    |                   |  {set dispatcher interrupt;
                |    |                   |   sysstack[%1277].<12>:=1}
                |    |                   |else sysstack[A+3]:=0;
                |    |                   |RP:=RP-1
                |    |                   | !!! Note !!!
                |    |                   | sysstack must be resident
0 0  0  1  0 reg|STRP|set RP              |RP:=reg
0 0  0  1  1 reg|STAR|store A in reg      |R[reg]:=A; RP:=RP-1
0 0  0  1  2 reg|NSAR|non-destructive     |R[reg]:=A
                |    |store A in reg      |
0 0  0  1  3 reg|LDRA|load register to A  |RP:=RP+1; cc(A:=R[reg])
0 0  0  1  4 reg|ADRA|add register to A   |ccn(A:=A+R[reg])
0 0  0  1  5 reg|SBRA|subtract register   |ccn(A:=A-R[reg])
                |    |from A              |
0 0  0  1  6 reg|ADAR|add A to register   |ccn(R[reg]:=R[reg]+A);
                |    |                   |RP:=RP-1
0 0  0  1  7 reg|SBAR|subtract A from     |ccn(R[reg]:=R[reg]-A);
                |    |register           |RP:=RP-1
0 0  0  2  0  0 |LADD|logical add         |ccl(B:=B+A); RP:=RP-1
0 0  0  2  0  1 |LSUB|logical subtract    |ccl(B:=B-A); RP:=RP-1
0 0  0  2  0  2 |LMPY|logical multiply    |cc(BA:=B'*'A); V:=0
0 0  0  2  0  3 |LDIV|logical divide      |V:=(C'>='A);
                |    |                   |(C,B):=(CB 'mod' A,CB'/'A);
                |    |                   |cc(B); RP:=RP-1
0 0  0  2  0  4 |LNEG|logical negate      |ccl(A:=-A)
0 0  0  2  0  5 |LCMP|logical compare     |cc(B':'A); RP:=RP-2
```

Table B-2. Instruction Definitions (Continued)

0	0	0	2	0	6*	FTL	find position in time list BA=time value	RP:=RP+1; BA:=CB; C:=sysstack[%107]; while C<>%107 do {BA:=BA-sysstack[C+2:C+3]; if < then done; C:=sysstack[C]} !!! Note !!! sysstack must be resident

0	0	0	2	0	7*	DTL	delete from time list
							A=element address
								a:=A; t:=sysstack[%107];
								RP:=RP+1;
								BA:=sysstack[t+2:t+3];
								while a <> t do
								{t:=sysstack[t];
								BA:=BA+sysstack[t+2:t+3]}
								!!! Note !!!
								sysstack must be resident

0	0	0	2	1	0	IADD	integer add	ccn(B:=B+A); RP:=RP-1
0	0	0	2	1	1	ISUB	integer subtract	ccn(B:=B-A); RP:=RP-1
0	0	0	2	1	2	IMPY	integer multiply	V:=~(-32768<=B*A<=32767); cc(B:=B*A); RP:=RP-1
0	0	0	2	1	3	IDIV	integer divide	V:=~(-32768<=B/A<=32767); cc(B:=B/A); RP:=RP-1
0	0	0	2	1	4	INEG	integer negate	ccn(A:=-A)
0	0	0	2	1	5	ICMP	integer compare	cc(B:A); RP:=RP-2
0	0	0	2	1	6* %	RPV	read PROM version numbers	RP:=RP+5; N:=0; Z:=1; CBA:=cs prom numbers D:=ept prom numbers E:=i/o channel prom number if i/o channel not available then {N:=1; Z:=0}
0	0	0	2	1	7*	TRCE	add an entry to the trace table EDCBA=entry	if TRBASE'<'TRLIM then {sysstack[TRACE:TRACE+4]:= EDCBA; TRACE:=TRACE+5; if TRACE'>'TRLIM then TRACE:=TRBASE}; RP:=RP-5
0	0	0	2	2	0	DADD	double add	ccn(DC:=DC+BA); RP:=RP-2
0	0	0	2	2	1	DSUB	double subtract	ccn(DC:=DC-BA); RP:=RP-2
0	0	0	2	2	2	DMPY	double multiply	ccn(DC:=DC*BA); RP:=RP-2
0	0	0	2	2	3	DDIV	double divide	ccn(DC:=DC/BA); V:= BA=0; RP:=RP-2
0	0	0	2	2	4	DNEG	double negate	ccn(BA:=-BA)
0	0	0	2	2	5	DCMP	double compare	cc(DC:BA); RP:=RP-4
0	0	0	2	2	6	MNGG	move words while not duplicate D=destination C=source B=count A=value<>to value of source	while cc(B)<>"=" and stack[C]<>A do {A:=stack[D]:=stack[C]; D:=D+1; C:=C+1; B:=B-1}; RP:=RP-1
0	0	0	2	2	7	MNDX	move words while not duplicate FE=destination DC=source B=count A=value<>to value of source	while cc(B)<>"=" and xmem[DC]<>A do {A:=xmem[FE]:=xmem[DC]; FE:=FE+2; DC:=DC+2; B:=B-1}; RP:=RP-1

Table B-2. Instruction Definitions (Continued)

0	0	0	2	3	0xx	QST	quad store	adr:=(if I=%230 then 0 else R[I.<14:15>+4])*4+A; stack[adr:adr+3]:=EDCB; RP:=RP-5
0	0	0	2	3	4xx	QLD	quad load	adr:=(if I=%234 then 0 else R[I.<14:15>+4])*4+A; RP:=RP+3; cc(DCBA:=stack[adr:adr+3])
0	0	0	2	4	0	QADD	quad add	ccn(HGFE:=HGFE + DCBA); RP:=RP-4
0	0	0	2	4	1	QSUB	quad subtract	ccn(HGFE:=HGFE - DCBA); RP:=RP-4
0	0	0	2	4	2$	QMPY	quad multiply	V:=if -2**63<=HGFE*DCBA<=2**63-1 then 0 else 1; HGFE:=HGFE * DCBA; cc(HGFE); RP:=RP-4
0	0	0	2	4	3$	QDIV	quad divide	V:=if DCBA=0 then 1 else 0; HGFE:=HGFE / DCBA; cc(HGFE); RP:=RP-4
0	0	0	2	4	4$	QNEG	quad negate	DCBA:=-DCBA; ccn(DCBA)
0	0	0	2	4	5$	QCMP	quad compare	cc(HGFE:DCBA)
0	0	0	2	4	6$	CQL	convert quad to logical	V:=if 0 <= DCBA <=2**16-1 then 0 else 1; D:=A; RP:=RP-3
0	0	0	2	4	7$	CQD	convert quad to double	V:=if -2**31 <=DCBA<= 2**31-1 then 0 else 1; DC:=BA; RP:=RP-2
0	0	0	2	5	nn0	QUP	quad scale up	DCBA:=DBCA* 10**(I.<13:14>+1); V:=if -2**63<=DCBA<=2**63-1 then 0 else 1; cc(DCBA)
0	0	0	2	5	nn1	QDWN	quad scale down	DCBA:=DBCA/ 10**(I.<13:14>+1); V:=0; cc(DCBA);

0	0	0	2	6	0$	CQA	convert quad to ASCII	cc(FEDC);
								B:=B+A;
								while A<>0 do
								{B:=B-1;
								bytedest(B):=
								%60+abs(FEDC) mod 10;
								FEDC:=FEDC/10;
								A:=A-1}
								V:=if FEDC=0 then 0 else 1;
								RP:=RP-6

Table B-2. Instruction Definitions (Continued)

0	0	0	2	6	1$	CAQV	convert ASCII to quad with initial value	V:=0;
								N:=1;
								while E<>0 and V=0 and N=1 do
								{ccb(t:=bytedest(F));
								if N=1 then
								{DCBA:=DCBA*10 + t&%17;
								V:=if DCBA<=2**63-1
								then 0 else 1;
								F:=F+1;
								E:=E-1}}
								cc(E) !cce if entire string
								!is ASCII digits.
								!ccg if not.
								!Note: initial value (DCBA)
								! should be positive.

0	0	0	2	6	2$	CAQ	convert ASCII to quad	RP:=RP+4;
								DCBA:=0;
								V:=0;
								N:=1;
								while E<>0 and V=0 and N=1 do
								{ccb(t:=bytedest(F));
								if N=1 then
								{DCBA:=DCBA*10 + t&%17;
								V:=if DCBA<=2**63-1
								then 0 else 1;
								F:=F+1;
								E:=E-1}}
								cc(E) !cce if entire string
								!is ASCII digits.
								!ccg if not.

0	0	0	2	6	3$	QRND	quad round	DCBA:=(if DCBA<0 then DCBA-5 else DCBA+5) / 10; V:=0; cc(DCBA)
0	0	0	2	6	4$	CQI	convert quad to integer	V:=if -2**15 <=DCBA<= 2**15-1 then 0 else 1; D:=A; RP:=RP-3;
0	0	0	2	6	5$	CDQ	convert double to quad	(t,u):=BA; s:=if B<0 then %177777 else 0; RP:=RP+2; DCBA:=(s,s,t,u)
0	0	0	2	6	6$	CIQ	convert integer to quad	t:=A; s:=if A<0 then %177777 else 0; RP:=RP+3; DCBA:=(s,s,s,t)
0	0	0	2	6	7$	CLQ	convert logical to quad	t:=A;RP:=RP+3; DCBA:=(0,0,0,t)

Table B-2. Instruction Definitions (Continued)

```
0 0  0  2  7  0#|FADD|floating add      |t1:=exponent(C);
                |     |DC:=DC+BA        |t2:=exponent(A);
                |     |                 |if BA<>0 and DC<>0
                |     |                 | and abs(t1-t2)<24 then
                |     |                 |  {sign1:=D.<0>;
                |     |                 |   sign2:=B.<0>;
                |     |                 |   D.<0>:=B.<0>:=1;
                |     |                 |   exponent(C):=0;
                |     |                 |   exponent(A):=0;
                |     |                 |   s:=t1-t2;
                |     |                 |   if s>=0 then
                |     |                 |     BA:=BA'>>'s;
                |     |                 |   else
                |     |                 |    {DC:=DC'>>'-s;
                |     |                 |     DC:=:BA;
                |     |                 |     t1:=t2}
                |     |                 |   if sign1=sign2 then
                |     |                 |    {DC:=DC'+'BA;
                |     |                 |     if carry then
                |     |                 |      {DC:=DC'>>'1;
                |     |                 |       t1:=t1+1;
                |     |                 |       D.<0>:=1}}
                |     |                 |   else
                |     |                 |    {DC:=DC'-'BA;
                |     |                 |     if not carry then
                |     |                 |      {DC:=-DC;
                |     |                 |       sign1:=~sign1}
                |     |                 |     if DC=0 then
                |     |                 |       t1:=sign1:=0
                |     |                 |     else
                |     |                 |       while D.<0>=0 do
                |     |                 |         {DC:=DC'<<'1;
                |     |                 |          t1:=t1-1}}
                |     |                 |   DC:=DC'+'%400;
                |     |                 |   if carry then
                |     |                 |      t1:=t1+1;
                |     |                 |   if t1.<6>=1 then
                |     |                 |      call overflow;
                |     |                 |   D.<0>:=sign1;
                |     |                 |   exponent(C):=t1}
                |     |                 |else
                |     |                 |   if DC=0 or t1-t2<=-24 then
                |     |                 |     DC:=BA;
                |     |                 |cc(DC); RP:=RP-2
0 0  0  2  7  1#|FSUB|floating subtract |if BA<>0 then
                |     |DC:=DC-BA        |  B.<0>:=~B.<0>;
                |     |                 |goto FADD
```

Table B-2. Instruction Definitions (Continued)

0	0	0	2	7	2#	FMPY	floating multiply
							DC:=DC*BA
								if DC=0 or BA=0 then
								  DC:=0
								else
								  {t1:=exponent(C);
								  t2:=exponent(A);
								  exp:=t1+t2-255;
								  sign:=D.<0> xor B.<0>;
								  D.<0>:=B.<0>:=1;
								  exponent(C):=0;
								  exponent(A):=0;
								  DCBA:=DC'*'BA;
								  norm(DC);
								  DC:=DC'+'%400;
								  if carry then
								  exp:=exp+1;
								  if exp.<6>=1 then
								  call overflow;
								  D.<0>:=sign;
								  exponent(C):=exp}
								cc(DC); RP:=RP-2

0	0	0	2	7	3#	FDIV	floating divide
							DC:=DC/BA
								if BA=0 then
								  call overflow;
								if DC<>0 then
								  {t1:=exponent(C);
								  t2:=exponent(A);
								  exp:=t1-t2+256;
								  sign:=D.<0> xor B.<0>;
								  D.<0>:=B.<0>:=1;
								  exponent(C):=0;
								  exponent(A):=0;
								  DC:=DC'/'BA;
								  norm(DC);
								  DC:=DC'+'%400;
								  if carry then
								  exp:=exp+1;
								  if exp.<6>=1 then
								  call overflow;
								  D.<0>:=sign;
								  exponent(C):=exp}
								cc(DC); RP:=RP-2

0	0	0	2	7	4#	FNEG	floating negate BA:=-BA	if BA<>0 then B.<0>:=~B.<0>; cc(BA)

0	0	0	2	7	5#	FCMP	floating compare
							DC:BA
								if D.<0> <> B.<0> then
								  cc(D:B)
								else
								  {sign:=D.<0>;
								  D.<0>:=B.<0>:=0;
								  t1:=exponent(C);
								  t2:=exponent(A);
								  if t1<>t2 then
								  if sign=0 then
								  cc(t1:t2)
								  else cc(t2:t1)
								  else
								  if sign=0 then
								  cc(DC:BA)
								  else cc(BA:DC)}
								RP:=RP-4

0	0	0	2	7	6#	CEF	convert extended to floating	exponent(C):=exponent(A); RP:=RP-2

Table B-2. Instruction Definitions (Continued)

0	0	0	2	7	7#	CEFR	convert extended to floating with rounding	sign:=D.<0>; D.<0>:=1;
								exp:=exponent(A);
								DC:=DC'+'%400;
								if carry then
								{exp:=exp+1;
								if exp.<6> then V:=1}
								D.<0>:=sign;
								exponent(C):=exp;
								RP:=RP-2

0	0	0	3	0	0#	EADD	extended add
							HGFE:=HGFE+DCBA
								t1:=exponent(E);
								t2:=exponent(A);
								if DCBA<>0 and HGFE<>0
								and abs(t1-t2)<56 then
								{sign1:=H.<0>;
								sign2:=D.<0>;
								H.<0>:=D.<0>:=1;
								exponent(E):=0;
								exponent(A):=0;
								s:=t1-t2;
								if s>=0 then
								DCBA:=DCBA'>>'s;
								else
								{HGFE:=HGFE'>>'-s;
								HGFE:=:DCBA;
								t1:=t2}
								if sign1=sign2 then
								{HGFE:=HGFE'+'DCBA;
								if carry then
								{HGFE:=HGFE'>>'1;
								t1:=t1+1;
								H.<0>:=1}}
								else
								{HGFE:=HGFE'-'DCBA;
								if not carry then
								{HGFE:=-HGFE;
								sign1:=~sign1}
								if HGFE=0 then
								t1:=sign1:=0
								else
								while H.<0>=0 do
								{HGFE:=HGFE'<<'1;
								t1:=t1-1}}
								HGFE:=HGFE'+'%400;
								if carry then
								t1:=t1+1;
								if t1.<6>=1 then
								call overflow;
								H.<0>:=sign1;
								exponent(E):=t1}
								else
								if HGFE=0 or t1-t2<=-56
								then HGFE:=DCBA;
								cc(HGFE); RP:=RP-4

0	0	0	3	0	1#	ESUB	extended subtract HGFE:=HGFE-DCBA	if DCBA<>0 then D.<0>:=~D.<0>; goto EADD

Table B-2. Instruction Definitions (Continued)

0	0	0	3	0	2#	EMPY	extended multiply
							HGFE:=HGFE*DCBA
								if HGFE=0 or DCBA=0 then
								  HGFE:=0
								else
								  {t1:=exponent(E);
								  t2:=exponent(A);
								  exp:=t1+t2-255;
								  sign:=H.<0> xor D.<0>;
								  H.<0>:=D.<0>:=1;
								  exponent(E):=0;
								  exponent(A):=0;
								  HGFE:=HGFE'*'DCBA;
								  norm(HGFE);
								  HGFE:=HGFE'+'%400;
								  if carry then
								  exp:=exp+1;
								  if exp.<6>=1 then
								  call overflow;
								  H.<0>:=sign;
								  exponent(E):=exp}
								cc(HGFE); RP:=RP-4

0	0	0	3	0	3#	EDIV	extended divide
							HGFE:=HGFE/DCBA
								if DCBA=0 then
								  call overflow;
								if HGFE<>0 then
								  {t1:=exponent(E);
								  t2:=exponent(A);
								  exp:=t1-t2+256;
								  sign:=H.<0> xor D.<0>;
								  H.<0>:=D.<0>:=1;
								  exponent(E):=0;
								  exponent(A):=0;
								  HGFE:=HGFE'/'DCBA;
								  norm(HGFE);
								  HGFE:=HGFE'+'%400;
								  if carry then
								  exp:=exp+1;
								  if exp.<6>=1 then
								  call overflow;
								  H.<0>:=sign;
								  exponent(E):=exp}
								cc(HGFE); RP:=RP-4

0	0	0	3	0	4#	ENEG	extended negate DCBA:=-DCBA	if DCBA<>0 then D.<0>:=~D.<0>; cc(DCBA)

0	0	0	3	0	5#	ECMP	extended compare
							HGFE:DCBA
								if H.<0> <> D.<0> then
								  cc(H:D)
								else
								  {sign:=H.<0>;
								  H.<0>:=D.<0>:=0;
								  t1:=exponent(E);
								  t2:=exponent(A);
								  if t1<>t2 then
								  if sign=0 then
								  cc(t1:t2)
								  else cc(t2:t1)
								  else
								  if sign=0 then
								  cc(HGFE:DCBA)
								  else cc(DCBA:HGFE)}

Table B-2. Instruction Definitions (Continued)

0	0	0	3	0	6#	CDF	convert double to floating	sign:=B.<0>; exp:=31+256;
								if sign=1 then BA:=-BA;
								if BA<>0 then
								{norm(BA);
								exponent(A):=exp;
								B.<0>:=sign}

0	0	0	3	0	7	CDI	convert double to integer	if B+A.<0> <> 0 then V:=1 else V:=0; B:=A; RP:=RP-1

0	0	0	3	1	0#	CFIR	convert floating to integer with rounding	t:=15+256-exponent(A);
								sign:=B.<0>;
								if -2**15 <= BA <= 2**15-1
								then {B.<0>:=1;
								BA:=BA'>>'t;
								BA:=BA'+'%100000;
								if sign=1 then B:=-B
								else if B.<0>=1 then
								V:=1}
								else V:=1;
								cc(B); RP:=RP-1

0	0	0	3	1	1#	CFI	convert floating to integer	t:=15+256-exponent(A);
								sign:=B.<0>;
								if -2**15 <= BA <= 2**15-1
								then {B.<0>:=1;
								BA:=BA'>>'t;
								if sign=1 then B:=-B}
								else V:=1;
								cc(B); RP:=RP-1

0	0	0	3	1	2#	CFD	convert floating to double	t:=31+256-exponent(A);
								sign:=B.<0>;
								if -2**31 <= BA <= 2**31-1
								then {B.<0>:=1;
								exponent(A):=0;
								BA:=BA'>>'t;
								if sign=1 then
								BA:=-BA}
								else V:=1;
								cc(BA)

0	0	0	3	1	3#	CFDR	convert floating to double with rounding	t:=31+256-exponent(A);
								sign:=B.<0>;
								if -2**31 <= BA <= 2**31-1
								then {B.<0>:=1;
								exponent(A):=0;
								BAs:=BAs'>>'t;
								BAs:=BAs'+'%100000;
								if sign=1 then
								BA:=-BA
								else if B.<0>=1 then
								V:=1}
								else V:=1;
								cc(BA)

0	0	0	3	1	4#	CED	convert extended to double	t:=31+256-exponent(A);
								sign:=D.<0>;
								if -2**31 <= DCBA <= 2**31-1
								then {D.<0>:=1;
								DC:=DC'>>'t;
								if sign=1 then
								DC:=-DC}
								else V:=1;
								cc(DC); RP:=RP-2

Table B-2. Instruction Definitions (Continued)

0	0	0	3	1	5#	CEDR	convert extended to double with rounding	t:=31+256-exponent(A);
								sign:=D.<0>;
								if -2**31 <= DCBA <= 2**31-1
								 then {D.<0>:=1;
								 DCB:=(DCB'>>'t)
								 '+'%100000;
								 if sign=1 then
								 DC:=-DC
								 else if D.<0>=1 then
								 V:=1}
								 else V:=1;
								cc(DC); RP:=RP-2

0	0	0	3	1	6#	CEIR	convert extended to integer with rounding	t:=15+256-exponent(A);
								sign:=D.<0>;
								if -2**15 <= DCBA <= 2**15-1
								 then {D.<0>:=1;
								 DC:=(DC'>>'t)
								 '+'%100000;
								 if sign=1 then D:=-D
								 else if D.<0>=1 then
								 V:=1}
								 else V:=1;
								cc(D); RP:=RP-3

0	0	0	3	1	7#	IDXD	calculate index offset and test indices for bounds violation
							
							(bounds table in data space)
								t:=stack[A];
								bc:=t.<0>; t.<0>:=0;
								indv:=0; psize:=1;
								s:=A;
								while t>0 do
								 {lower:=stack[s:=s+1];
								 upper:=stack[s:=s+1];
								 if B<lower and bc=0 then
								 {V:=1; t =0;
								 cc(-1); R[7]:=B}
								 if B>upper and bc=0 then
								 {V:=1; t =0;
								 cc(1); R[7]:=B}
								 size:=upper-lower+1;
								 B:=B-lower;
								 indv:=indv+psize*B;
								 psize:=psize*size;
								 RP:=RP-1; t:=t-1}
								if V=0 then
								 {R[7]:=indv;
								 cc(R[7])}
								RP:=RP-1

0	0	0	3	2	0#	CFQ	convert floating to quad	t:=63+256-exponent(A);
								sign:=B.<0>; RP:=RP+2;
								if -2**63 <= DC <= 2**63-1
								 then {D.<0>:=1;
								 exponent(C):=0;
								 B:=A:=0;
								 DCBA:=DCBA'>>'t;
								 if sign=1 then
								 DCBA:=-DCBA}
								 else V:=1;
								cc(DCBA)

Table B-2. Instruction Definitions (Continued)

0	0	0	3	2	1#	CFQR	convert floating to quad with rounding	t:=63+256-exponent(A);
								sign:=B.<0>; RP:=RP+2;
								if -2**63 <= DC <= 2**63-1
								then {D.<0>:=1;
								exponent(C):=0;
								B:=A:=s:=0;
								DCBAs:=(DCBAs'>>'t)
								'+'%100000;
								if sign=1 then
								DCBA:=-DCBA}
								else V:=1;
								cc(DCBA)

0	0	0	3	2	2#	CEQ	convert extended to quad	t:=63+256-exponent(A);
								sign:=D.<0>;
								if -2**63 <= DCBA <= 2**63-1
								then {D.<0>:=1;
								exponent(A):=0;
								DCBA:=DCBA'>>'t;
								if sign=1 then
								DCBA:=-DCBA}
								else V:=1;
								cc(DCBA)

0	0	0	3	2	3#	CEQR	convert extended to quad with rounding	t:=63+256-exponent(A);
								sign:=D.<0>;
								if -2**63 <= DCBA <= 2**63-1
								then {D.<0>:=1;
								exponent(A):=0;
								s:=0;
								DCBAs:=(DCBAs'>>'t)
								'+'%100000;
								if sign=1 then
								DCBA:=-DCBA}
								else V:=1;
								cc(DCBA)

0	0	0	3	2	4#	CQF	convert quad to floating	sign:=D.<0>; exp:=63+256;
								if sign=1 then
								DCBA:=-DCBA;
								if DCBA<>0 then
								{norm(DCBA);
								exponent(C):=exp;
								D.<0>:=sign}
								RP:=RP-2

0	0	0	3	2	5#	CFE	convert floating to extended	G:=exponent(A); exponent(A):=0; H:=0; RP:=RP+2

0	0	0	3	2	6#	CDFR	convert double to floating with rounding	sign:=B.<0>; exp:=31+256;
								if sign=1 then
								BA:=-BA;
								if BA<>0 then
								{norm(BA);
								BA:=BA'+'%400;
								if carry then
								exp:=exp+1;
								exponent(A):=exp;
								B.<0>:=sign}

0	0	0	3	2	7	CID	convert integer to double	H:=A; A := A>>15; V:=0; RP:=RP+1

Table B-2. Instruction Definitions (Continued)

0	0	0	3	3	0#	CQFR	convert quad to floating with rounding	sign:=D.<0>; exp:=63+256;
								if sign=1 then
								DCBA:=-DCBA;
								if DCBA<>0 then
								{norm(DCBA);
								DC:=DC'+'%400;
								if carry then
								exp:=exp+1;
								exponent(C):=exp;
								D.<0>:=sign}
								RP:=RP-2

0	0	0	3	3	1#	CIF	convert integer to floating	sign:=A.<0>; exp:=15+256;
								if sign=1 then A:=-A;
								if A<>0 then
								{norm(A);
								H:=exp;
								A.<0>:=sign}
								else H:=0;
								RP:=RP+1

0	0	0	3	3	2#	CIE	convert integer to extended	sign:=A.<0>; exp:=15+256;
								if sign=1 then A:=-A;
								H:=G:=0;
								if A<>0 then
								{norm(A);
								F:=exp;
								A.<0>:=sign}
								else F:=0;
								RP:=RP+3

0	0	0	3	3	3	XSMX	checksum extended block D=initial checksum CB=block address A=count	while A<>0 do {D:=D xor xmem[CB]; A:=A-1; CB:=CB+2}; RP:=RP-3

0	0	0	3	3	4#	CDE	convert double to extended	sign:=B.<0>; exp:=31+256;
								if sign=1 then BA:=-BA;
								H:=0;
								if BA<>0 then
								{norm(BA);
								G:=exp;
								B.<0>:=sign}
								else G:=0;
								RP:=RP+2

0	0	0	3	3	5#	CQER	convert quad to extended with rounding	sign:=D.<0>; exp:=63+256;
								if sign=1 then
								DCBA:=-DCBA;
								if DCBA<>0 then
								{norm(DCBA);
								DCBA:=DCBA'+'%400;
								if carry then
								exp:=exp+1;
								exponent(A):=exp;
								D.<0>:=sign}

0	0	0	3	3	6#	CQE	convert quad to extended	sign:=D.<0>; exp:=63+256;
								if sign=1 then
								DCBA:=-DCBA;
								if DCBA<>0 then
								{norm(DCBA);
								exponent(A):=exp;
								D.<0>:=sign}

Table B-2. Instruction Definitions (Continued)

0	0	0	3	3	7#	CEI	convert extended to integer	t:=15+256-exponent(A);
								sign:=D.<0>;
								if -2**15 <= DCBA <= 2**15-1
								then {D.<0>:=1;
								D:=D'>>'t;
								if sign=1 then D:=-D}
								else V:=1;
								cc(D); RP:=RP-3

0	0	0	3	4	0			*** undefined ***
0	0	0	3	4	1			*** undefined ***
0	0	0	3	4	2	LWUC	load word from user code space	cc(A:=mem[2,A])
0	0	0	3	4	3	XSMG	checksum block C=initial checksum B=block address A=count	while A<>0 do {C:=C xor stack[B]; A:=A-1; B:=B+1}; RP:=RP-2

0	0	0	3	4	4#	IDX1	calculate index offset and test index bounds for 1 dimension
							
							(bounds table in code space)
								lower:=code[A];
								upper:=code[A+1];
								if B<lower then
								{V:=1; cc(-1);
								R[7]:=B}
								if B>upper then
								{V:=1; cc(1);
								R[7]:=B}
								if V=0 then
								{R[7]:=B-lower;
								cc(R[7])}
								RP:=RP-2

0	0	0	3	4	5#	IDX2	calculate index offset and test index bounds for 2 dimensions
							
							(bounds table in code space)
								lower:=code[A];
								upper:=code[A+1];
								if B<lower then
								{V:=1; cc(-1);
								R[7]:=B}
								if B>upper then
								{V:=1; cc(1);
								R[7]:=B}
								s:=upper-lower+1;
								B:=B-lower;
								lower:=code[A+2];
								upper:=code[A+3];
								if C<lower then
								{V:=1; cc(-1);
								R[7]:=C}
								if C>upper then
								{V:=1; cc(1);
								R[7]:=C}
								if V=0 then
								{R[7]:=(C-lower)*s+B;
								cc(R[7])}
								RP:=RP-3

Table B-2. Instruction Definitions (Continued)

```
0 0  0  3  4  6#|IDX3|calculate index   |indv:=0; psize:=1;
                |    |offset and test   |for i=1 to 3 by 1 do
                |    |index bounds      | {lower:=code[A];
                |    |for 3 dimensions  |  upper:=code[A:=A+1];
                |    |                  |  if B<lower then
                |    |(bounds table     |     {V:=1;
                |    |in code space)    |      cc(-1); R[7]:=B}
                |    |                  |  if B>upper then
                |    |                  |     {V:=1;
                |    |                  |      cc(1); R[7]:=B}
                |    |                  |  size:=upper-lower+1;
                |    |                  |  B:=B-lower;
                |    |                  |  indv:=indv+psize*B;
                |    |                  |  psize:=psize*size;
                |    |                  |  B:=A+1;
                |    |                  |  RP:=RP-1}
                |    |                  |if V=0 then
                |    |                  |  {R[7]:=indv;
                |    |                  |   cc(R[7])}
                |    |                  |RP:=RP-1
0 0  0  3  4  7#|IDXP|calculate index   |t:=code[A];
                |    |offset and test   |bc:=t.<0>; t.<0>:=0;
                |    |indices for       |indv:=0; psize:=1;
                |    |bounds violation  |s:=A;
                |    |                  |while t>0 do
                |    |(bounds table     | {lower:=code[s:=s+1];
                |    |in code space)    |  upper:=code[s:=s+1];
                |    |                  |  if B<lower and bc=0 then
                |    |                  |     {V:=1; t:=0;
                |    |                  |      cc(-1); R[7]:=B}
                |    |                  |  if B>upper and bc=0 then
                |    |                  |     {V:=1; t:=0;
                |    |                  |      cc(1); R[7]:=B}
                |    |                  |  size:=upper-lower+1;
                |    |                  |  B:=B-lower;
                |    |                  |  indv:=indv+psize*B;
                |    |                  |  psize:=psize*size;
                |    |                  |  RP:=RP-1; t:=t-1}
                |    |                  |if V=0 then
                |    |                  |  {R[7]:=indv;
                |    |                  |   cc(R[7])}
                |    |                  |RP:=RP-1
0 0  0  3  5  0 |LWAS|load SG word via A|cc(A:=dest(A))
0 0  0  3  5  1 |SWAS|stor SG word via A|dest(A):=B; RP:=RP-2
0 0  0  3  5  2 |LDAS|load SG double    |RP:=RP+1;
                |    |via A             |cc(BA:=dest(B:B+1))
0 0  0  3  5  3 |SDAS|store SG double   |dest(A:A+1):=CB;
                |    |via A             |RP:=RP-3;
0 0  0  3  5  4 |LBAS|load SG byte via A|ccb(A:=bytedest(A))
0 0  0  3  5  5 |SBAS|store SG byte     |bytedest(A):=B;
                |    |via A             |RP:=RP-2
0 0  0  3  5  6 |CDX |count duplicate   |while B<>0 and
                |    |words extended    | xmem[DC]=xmem[DC-2] do
                |    |DC=buffer address | {A:=A+1;
                |    |B=buffer size     |  B:=B-1;
                |    |A=duplicate count |  DC:=DC+2}
0 0  0  3  5  7 |DFS |deposit field in  |cc(dest(A):=(dest(A) & ~B)
                |    |SG memory         |                | (C & B));
                |    |                  |RP:=RP-3
0 0  0  3  6  0 |LWA |load word via A   |cc(A:=stack[A])
0 0  0  3  6  1 |SWA |store word via A  |stack[A]:=B;  RP:=RP-2
```

Table B-2. Instruction Definitions (Continued)

0	0	0	3	6	2		LDA	load double via A	RP:=RP+1; cc(BA:=stack[B:B+1])
0	0	0	3	6	3		SDA	store double via A	stack[A:A+1]:=CB; RP:=RP-3;
0	0	0	3	6	4		LBA	load byte via A	ccb(A:=bytedest(A))
0	0	0	3	6	5		SBA	store byte via A	bytedest(A):=B; RP:=RP-2
0	0	0	3	6	6		CDG	count duplicate words C=buffer address B=buffer size A=duplicate count	while B<>0 and stack[C]=stack[C-1] do {A:=A+1; B:=B-1; C:=C+1}
0	0	0	3	6	7		DFG	deposit field in memory	cc(stack[A]:=(stack[A] & ~B) | (C & B)); RP:=RP-3
0	0	0	3	7	0				
			.						*** undefined ***
			.						
0	0	0	3	7	7				

0	0	0	4	0	0	*	WWCS	write LCS
								D=LCS address
								C=buffer map
								B=buffer address
								A=ucode word count
									while A>0 do
									{LCS[D]:=mem[C,B]^mem[C,B+1]
									^mem[C,B+2].<0:3>;
									if (A:=A-1)=0
									then goto done;
									D:=D+1;B:=B+2;
									LCS[D]:=mem[C,B].<8:15>
									^mem[C,B+1]
									^mem[C,B+2].<0:11>;
									D:=D+1; B:=B+3; A:=A-1;
									};
									done: N:=0; Z:=1; RP:=RP-4
									!!! Note !!!
									all memory referenced must be present

0	0	0	4	0	0	*	WWCS	write HCS/VCS/EPT
								D=HCS/VCS/EPT address
								C.<0:1>=control store type:
								00=HCS
								01=VCS
								10=EPT/ECS
								11=reserved
								C.<2:15>^B=abs. extended buffer address
								A=ucode count
									if UCOPTIONFLAG=-1 then
									{N:=0; Z:=0; goto done2};
									y:=1; x:=case C.<0:1> of
									{12;6:8;0};
									while A > 0 do
									{case C.<0:1> of
									{{HCS[D]:=xmem[CB:CB+5]};
									{VCS[D]:=xmem[CB:CB+2]};
									{EPT[D]^ECS[D]:=
									xmem[CB:CB+3];
									if D.<0:6><>0 then y:=64};
									{};
									};
									if (A:=A-1)=0
									then goto done;
									D:=D+y; CB:=CB+x;
									};
									done: N:=0; Z:=1;
									done2: RP:=RP-4
									Note
									VCS/HCS addresses should not exceed 2**13-1

Table B-2. Instruction Definitions (Continued)

0	0	0	4	0	1*
					%
						VWCS	verify LCS
							D=LCS address
							C=buffer map
							B=buffer address
							A=ucode word count
								N:=0;Z:=1;
								while Z and A>0 do
								{if LCS[D]<>mem[C,B]
								^mem[C,B+1]
								^mem[C,B+2].<0:3>
								then {N:=1;Z:=0};
								if N or (A:=A-1)=0
								then goto done;
								D:=D+1;B:=B+2;
								if LCS[D]<>mem[C,B].<8:15>
								^mem[C,B+1]
								^mem[C,B+2].<0:11>
								then {N:=1;Z:=0}
								else {D:=D+1;B:=B+3;A:=A-1};
								};
								done: RP:=RP-4
								!!! Note !!!
								all memory referenced must be present
								bus packets may not be received correctly while a VWCS is executing

0	0	0	4	0	1*
					&
						VWCS	verify HCS/VCS/EPT
							
							D=HCS/VCS/EPT address
							C.<0:1>=control store type:
							00=HCS
							01=VCS
							10=EPT/ECS
							11=reserved
							C.<2:15>^B=abs. extended buffer address
							A=ucode count
								if UCOPTIONFLAG=-1 then
								{N:=0; Z:=0; goto done};
								N:=0; Z:=1; y:=1; x:=
								case C.<0:1> of {12;6;8;0};
								while Z and A > 0 do
								{case C.<0:1> of
								{{if HCS[D]<>
								xmem[CB:CB+5] then
								{N:=1; Z:=0; goto done}};
								{if VCS[D]<>
								xmem[CB:CB+2] then
								{N:=1; Z:=0; goto done}};
								{if EPT[D]^ECS[D]<>
								xmem[CB:CB+3] then
								{N:=1; Z:=0; goto done};
								if D<0:6><>0 then y:=64};
								{};
								};
								if N or (A:=A-1)=0
								then goto done;
								D:=D+y; CB:=CB+x;
								};
								done: RP:=RP-4
								Note
								VCS/HCS addresses should not exceed 2**13-1

0	0	0	4	0	2*
					%
						RWCS	read LCS
							D=LCS address
							C=buffer map
							B=buffer address
							A=ucode word count
								while A>0 do
								{mem[C,B]^mem[C,B+1]
								^mem[C,B+2].<0:3>:=LCS[D];
								if (A:=A-1)=0 then
								then goto done;
								D:=D+1;B:=B+2;
								mem[C,B].<8:15>^mem[C,B+1]^
								mem[C,B+2].<0:11>:=LCS[D];
								D:=D+1;B:=B+3;A:=A-1};
								done: RP:=RP-4
								!!! Note !!!
								all memory referenced must be present

Table B-2. Instruction Definitions (Continued)

```
0 0  0  4  0  2*|RWCS|read WCS or EPT    |if UCOPTIONALFLAG=-1 then
                &|    |                  |{N:=0; Z:=0; goto done};
                 |    |                  |N:=0; Z:=1; y:=1; x:=
                 |    |                  | case C.<0:1> of
                 |    |D=HCS/VCS/EPT     |  {12;6;8;0};
                 |    |  address         |while A > 0 do
                 |    |C.<0:1>=control   |{case C.<0:1> of
                 |    |  store type:     | {{xmem[CB:CB+5]:=HCS[D]};
                 |    |  00=HCS          |  {xmem[CB:CB+2]:=VCS[D]};
                 |    |  01=VCS          |  {xmem[CB:CB+3]:=
                 |    |  10=EPT/ECS      |    EPT[D]^ECS[D];
                 |    |  11=reserved     |   if D.<0:6><>0 then x:=64};
                 |    |C.<2:15>^B=abs.   |  {};
                 |    |  extended buffer | };
                 |    |  address         | if (A:=A-1)=0
                 |    |A=ucode count     | then goto done;
                 |    |                  | D:=D+y; CB:=CB+x;
                 |    |                  |};
                 |    |                  |done: RP:=RP-4
                 |    |                  |***Note***
                 |    |                  | VCS/HCS addresses should not
                 |    |                  | exceed 2**13-1
0 0  0  4  0  3  |    |                  | *** undefined ***
0 0  0  4  0  4*|SMBP|set memory brkpt  |breakpointmode:=B.<0:2>;
                %|    |B.<0>=read flag   |breakpointaddress:=
                 |    | .<1>=execute flag|              B.<9:15>^A;
                 |    | .<2>=write flag  |BPADDR:=BA; RP:=RP-2;
                 |    | .<9:15>=high-    | !!! Note !!!
                 |    |       order addr |the address is a physical
                 |    |A=low-order addr  |memory address
                 |    |                  |any and all combinations of
                 |    |                  |access flags may be set
                 |    |                  |BA=0D will disable the trap
0 0  0  4  0  4*|SMBP|set memory brkpt  |breakpointmode:=C.<0:2>;
                &|    |C.<0>=read flag   |breakpointaddress:=BA;
                 |    |C.<1>=execute flag|BPADDR:=CA;
                 |    |C.<2>=write flag  |BPADDRX:=B;
                 |    |C.<3:8>=mab type  |RP:=RP-3
                 |    |BA=extended addr  | !!! Note !!!
                 |    |   (absolute)     |any and all combinations of
                 |    |                  |access flags may be set
                 |    |                  |C=0 will disable the trap
                 |    |                  |all memory referenced must
                 |    |                  |be present
0 0  0  4  0  5*|FRST|firmware reset    |reset and stop instruction
                 |    |                  |execution
0 0  0  4  0  6  |LBX |load byte extended|ccb(B:=bxmem[BA]);RP:=RP-1
0 0  0  4  0  7  |SBX |store byte extnd. |bxmem[BA]:=C; RP:=RP-3
0 0  0  4  1  0  |LWX |load word extended|cc(B:=xmem[BA]);RP:=RP-1
0 0  0  4  1  1  |SWX |store word extnd. |xmem[BA]:=C; RP:=RP-3
0 0  0  4  1  2  |LDDX|load double extnd.|cc(BA:=xmem[BA:BA+3])
0 0  0  4  1  3  |SDDX|store dbl. extnd. |xmem[BA:BA+3]:=DC;RP:=RP-4
0 0  0  4  1  4  |LQX |load quad extended|RP:=RP+2;
                 |    |                  |cc(DCBA:=xmem[DC:DC+7])
0 0  0  4  1  5  |SQX |store quad        |xmem[BA:BA+7]:=FEDC;
                 |    |extended          |RP:=RP-6
0 0  0  4  1  6  |DFX |deposit field     |cc(xmem[BA]:=(xmem[BA] &
                 |    |extended          |              ~C | (D & C)));
                 |    |                  |RP:=RP-4;
```

Table B-2. Instruction Definitions (Continued)

0	0	0	4	1	7	MVBX	move bytes extended ED=destination address CB=source address A=byte count	while A<>0 do {bxmem[ED]:=bxmem[CB]; ED:=ED+1; CB:=CB+1; A:=A-1;}; RP:=RP-5;
0	0	0	4	2	0	MBXR	move bytes extended reverse ED=destination address CB=source address A=byte count	while A<>0 do {bxmem[ED]:=bxmem[CB]; ED:=ED-1; CB:=CB-1; A:=A-1;}; RP:=RP-5;
0	0	0	4	2	1	MBXX	move bytes extnd. and checksum F=initial xsum ED=destination address CB=source address A=byte count	while A<>0 do {bxmem[ED]:=t:=bxmem[CB]; F:=F xor t; ED:=ED+1; CB:=CB+1; A:=A-1;}; RP:=RP-5

0	0	0	4	2	2	CMBX	compare bytes
							extended
							ED=destination
							address
							CB=source address
							A=byte count
								N:=0; Z:=1;
								while Z and A<>0 do
								{cc(bxmem[ED]:bxmem[CB]);
								if Z then
								{A:=A-1;ED:=ED+1;
								CB:=CB+1;}};
								RP:=RP-5

0	0	0	4	2	3*	CRAX	convert rel. to abs. ext. address	if B.<0:14>=0 then {B.<0:14>:=CSSEG[DS]} else if B.<0:14>=1 then {B.<0:14>:=CSSEG[1]} else if B.<0:14>=2 then {B.<0:14>:=CSSEG[cmap]} else if B.<0:14>=3 then {B.<0:14>:=CSSEG[2]} else if B.<0>=0 then {if (BA+XL)->alu carry then { instruction failure }; BA:=BA+XB }; B.<0>:=1;

0	0	0	4	2	4*
					%
						RSPT	read segment page
							table entry
							BA=ext. address
								xa:=CRAX(BA);
								p:=xa.<15:20>;
								s:=xa.<2:14>;
								K:=0;
								if s >= SEGTABSIZE then
								{ B := 1; K := 1 }
								else
								if MAP[15,p mod 32+32]
								= s^p.<10> then
								{B:=MAP[15,p mod 32]}
								else
								{if p>=SEG[s*2].<9:15> then
								{B := 1; K := 1}
								else
								{if SEG[s*2].<0>=0 then
								B:=MAP[SEG[s*2].<0:4>,p]
								else
								B:=mem[SEG[s*2].<5:8>
								,SEG[s*2+1]+p] }
								};
								RP:=RP-1

Table B-2. Instruction Definitions (Continued)

0	0	0	4	2	4*
					&
						RSPT	read segment page table entry
							BA=ext. address
								xa:=CRAX(BA);
								p:=xa.<15:20>;
								s:=xa.<1:14>;
								K:=0;
								if s >= SEGTABSIZE or
								{ptmiss(xa) and
								p>=SEG[s*2].<9:15>} then
								{B:=1; K:=1;}
								else
								{if ptmiss(xa) then
								B:=mem[SEG[s*2].<5:8>,
								SEG[s*2+1]+p]
								else B:=PCACHE[s,p]};
								RP:=RP-1

0	0	0	4	2	5*
					%
						WSPT	write segment page table entry
							C=new spt entry
							BA=extended adrs.
								xa:=CRAX(BA);
								p:=xa.<15:20>;
								s:=xa.<2:14>;
								if s>=SEGTABLESIZE then
								{instruction failure};
								!Update cached entries
								if MAP[15,p mod 32+32]=
								s^p.<10> then
								{if C.<15> then
								MAP[15,p mod 32+32]:=-1
								else
								MAP[15,p mod 32]:=C
								};
								if p>=SEG[s*2].<9:15> then
								{instruction failure};
								!update mapped entries
								if SEG[s*2].<0>=0 then
								MAP[SEG[s*2].<0:4>,p]:=C;
								!unconditionally update
								!the page table
								mem[SEG[s*2].<5:8>,
								SEG[s*2+1]+p]:=C;
								RP:=RP-3.

0	0	0	4	2	5*
					&
						WSPT	write segment page table entry
							C=entry
							BA=ext. address
							(invalid on exit)
								xa:=CRAX(BA);
								p:=xa.<15:20>;
								s:=xa.<1:14>;
								if s >= SEGTABSIZE then
								Instruction Failure;
								PCACHE[s,p]:=C;
								if ~C.<13> then
								PCACHE.<15>:=1;
								PCACHETAG[s,p]:=
								(PCACHE[s,p]&%174003)|
								(xa.<2:10><<2);
								if p>=SEG[s*2].<9:15> then
								Instruction Failure;
								mem[SEG[s*2].<5:8>,
								SEG[s*2+1]+p]:=C;
								if C=1 then
								{xa.<16:31>:=
								xa.<16:31>&%174000;
								for i=0 to 127 do
								{if hit(xa) then
								invalidate entry;
								xa:xa+%20}};
								RP:=RP-3

Table B-2. Instruction Definitions (Continued)

							!!! Note !!! if R is not set in C, invalidate the entry; WSPT must not be used until SEG and the page tables are set up and SEGTABSIZE is present
0	0	0	4	2	6*	RXBL	read extended base and limit	RP:=RP+4; DCBA:=MAP[14,60:63]
0	0	0	4	2	7*	SXBL	set extended base and limit DC=base BA=limit	if (DC.<31> then { instruction failure }; XB := DC; XL := BA; RP:=RP-4

0	0	0	4	3	0*	LCKX	lock down extended
							memory
							D.<0>=lock only if
							already locked
							C=lock count
							BA=ext. address
								m:=RSPT(BA);
								p:=m.<0:12>;
								if m.<15>=0 and (D.<0>=0
								or PHYSEG[p]<0) then
								{if PHYSEG[p] < 0
								then
								{PHYSEG[p]:=PHYSEG[p]-C;
								K := 0}
								else
								{PHYSEG[p]:=-C;
								K := 1}
								Z:=1; N:=0}
								else {Z:=0; N:=1};
								RP:=RP-4

0	0	0	4	3	1*	ULKX	unlock extended
							memory
							D=map entry mask
							C=unlock count
							BA=ext. address
								m:=RSPT(xa:=CRAX(BA));
								p:=m.<0:12>;
								if m.<15>=0 and
								(x:=PHYSEG[p]+C)<=0 then
								{if x<>0 then PHYSEG[p]:=x
								else
								{PHYSEG[p]:=xa.<2:14>;
								WSPT(BA, m&D)};
								ccz(x)}
								else {Z:=0; N:=1};
								RP:=RP-4

0	0	0	4	3	2* %	CMRW	CME read/write B.<0:3>=map A=word address	N:=0;Z:=1; if I/O locked out then {mem[B.<0:3>,A] :=mem[B.<0:3>,A]; free I/O channel; if CME interrupt then Z:=0 else {N:=1; Z:=0}; RP:=RP-2

0	0	0	4	3	2*
					&
						CMRW	CME read/write
							BA=ext. address
								N:=0; Z:=1; x:=INTA;
								xmem[BA]:=xmem[BA];
								if CME interrupt then Z:=0;
								INTA:=x;
								RP:=RP-2
								!!! Note !!!
								Should read xmem[BA] from
								physical memory, not CACHE.

0	0	0	4	3	3			*** undefined ***
0	0	0	4	3	4			*** undefined ***
0	0	0	4	3	5			*** undefined ***
0	0	0	4	3	6*	RSMT	read from OSP	enable read from OSP
0	0	0	4	3	7*	WSMT	write to OSP	write first character to OSP

Table B-2. Instruction Definitions (Continued)

0	0	0	4	4	0* %	RIBA	read INTB and INTA registers	RP:=RP+2; B:=INTB; A:=INTA

0	0	0	4	4	1*
					%
						SVMP	save map entries	m:=word:=0;
								while word<%2000 do
								{memory[2,word]:=
								MAP[m.<12:15>,m.<0:5>]
								m:=m+%2000;
								if alu carry then m:=m+1;
								word:=word+1}

0	0	0	4	4	1* &	SVMP	save "map" entries	m:=0; do {memory[2,m]:= PCACHE[m.<6:9>,m.<10:15>]; m:=m+1} until m=%2000
0	0	0	4	4	2*	RPT	read process time	RP:=RP+2; BA:={if not DS then PTIME+(TIMER)+(10000* INTA.<13>) else PTIME}.

0	0	0	4	4	3*	SPT	set process timer	PTIME:=
								{if not DS then BA-TIMER-
								(INTA.<13>*10000)
								else
								BA
								};
								RP:=RP-2.

0	0	0	4	4	4	SCS	set code segment BA=byte address in current code	if ENV.CS=1 or ENV.LS=1 then B.<0:14>:=2 else B.<0:14>:=3;
0	0	0	4	4	5*	LQAS	load SG quad via A	RP:=RP+3; cc(DCBA:=sysstack[A:A+3])
0	0	0	4	4	6*	SQAS	store SG quad via A	sysstack[A:A+3]:=EDCB; RP:=RP-5
0	0	0	4	4	7* %	RCHN	reset I/O channel	if i/o channel available then {if A>=0 then channel ioreset else channel lockup at %0777; N:=0; Z:=1} else {N:=1; Z:=0}; RP:=RP-1
0	0	0	4	4	7* &	RCHN	reset I/O channel	if A>0 then channel ioreset else channel lockup with RPSA=%40; N:=0; Z:=1; RP:=RP-1

0	0	0	4	5	0*	BNDW	bounds test words
							C=word address in stack
							B=buffer size in words
							A=number of words of parameters and stack marker
								if A '>' L then
								cc(C:=1)
								else
								if B=0 or (C'<='L-A and
								C+B-1'<='L-A and C'<='C+B-1)
								or (C'>'L+350 and
								C'<='C+B-1 and
								(C+B-1).<0:5> <
								SEG[CSSEG[0]*2].<9:15>)
								then cc(C:=0)
								else cc(C:=1);
								RP:=RP-2

Table B-2. Instruction Definitions (Continued)

0	0	0	4	5	1%	BPT	instruction breakpoint trap	if BKPT = 0
								then interrupt via SIV #19
								BKPT := 0;
								i:=BPBASE;
								do
								{if sysstack[i]=CMSEG[cmap]
								and sysstack[i+1]=P-1
								then {I:=sysstack[i+2];
								roma:=EPT[I]};
								i:=i+BPSIZE}
								until i '>' BPLIM;
								Instruction failure

0	0	0	4	5	1&	BPT	instruction breakpoint trap	if BKPT = 0
								then interrupt via SIV #19
								BKPT := 0;
								i:=BPBASE;
								do
								{if sysstack[i]=SST[cseg]
								and sysstack[i+1]=P-1
								then {NI:=sysstack[i+2];
								NEXT INST};
								i:=i+BPSIZE}
								until i '>' BPLIM;
								Instruction failure

0	0	0	4	5	2*	BCLD	bus cold load	simulate a bus cold load from the panel
0	0	0	4	5	3* %	TPEF	test parity error freeze circuits	'Test parity circuits'; if error then {SD:=halt loop error code; halt }.

0	0	0	4	5	4	SCMP	set code map	if A.<0:6>=0 then
								{A.<0>:=CS;
								A.<1>:=LS;
								A.<2:6>:=CSPACEID
								}
								else
								if A.<0:6>=%133 then
								!external call
								{i:=SEG[CSSEG[cseg]*2]
								.<9:15>*%2000-1;
								A:=code[i-A.<7:15>]
								}.

0	0	0	4	5	5*
					&
						SRST	software reset	if cpu^type=TXP then
								{reload LCS from prom;
								if successful then
								cce
								else
								ccg
								}.

0	0	0	4	5	6*
					&
						DDTX	DDT request
							
							A.<8:15>=DDT function request
								if cpu^type=TXP then
								{if UREQ or ~TCBE then
								ccl
								else
								{issue function request(A)
								to DDT;
								cce
								};
								};
								RP:=RP-1.

Table B-2. Instruction Definitions (Continued)

0	0	0	4	5	7&	LIOC	load IOC entry A=subchannel #	IOCSPAD[A]':='IOC[A] for 4; RP:=RP-1
0	0	0	4	6	0&	SIOC	store IOC entry A=subchannel #	IOC[A]':='IOCSPAD[A] for 4; RP:=RP-1

0	0	0	4	6	2*	XIOC	exchange IOC entry
							
							A=subchannel #
							EDCB=IOC entry
								if i/o not locked out then
								cc(-1)
								else
								{temp ':=' IOC[A] for 4;
								IOC[A] ':=' EDCB for 4;
								if TNSII then
								{temp ':=' IOSPAD[A] for4;
								IOSPAD[A] ':=' EDCB for4;
								};
								free i/o channel;
								EDCB ':=' temp for 4;
								cc(0);
								};
								RP:=RP-1.

0	0	0	4	6	3*	SCPV	set current process variables A.<0:7>=ULseg size A.<8:15>=UC size B=UL seg base C=UC seg base	UC^BASE:=C; UL^BASE:=B; UC^SIZE:=A.<8:15>; UL^SIZE:=A.<0:7>; RP:=RP-3.

0	0	0	4	6	4*	BIKE	bicycle while idle	tests:='number of tests';
								while tests>=0 do
								{'perform cpu self test';
								if error then
								{SD:=error code;
								halt
								}
								tests:=tests-1;
								}.

0	0	0	4	6	5			*** undefined ***
0	0	0	4	6	6			*** undefined ***
0	0	0	4	6	7			*** undefined ***

0	0	0	4	7	0*	ASPT	Address of Segment
							Page Table header
							BA= extended addr.
							to convert
							C= byte offset
								xa:=CRAX(B,A);
								s:=xa.<2:14>;
								K:=0;
								if s>=SEGTABSIZE or
								SEG[t*2].<9:15> = 0 then
								{K:=1}
								else
								{xa.<0:14>:=SEG[t*2].<5:8>;
								xa.<15:31>:=SEG[t*2+1]*2;
								xa.<0>:=1;
								CB:=xa-$UDBL(C);
								};
								RP:=RP-1.

0	0	0	4	7	1	ESE	extensible stack
							expansion
								if (stack[L-3]+A)=0 then
								RP:=7
								else
								{cc(RP-1);
								call DPCL(sysstack[%171]);
								}.

0	0	0	4 . .	7	2			*** undefined ***
0	0	0	7	7	7			

Table B-2. Instruction Definitions (Continued)

0	0	1	-	-	-	CMPI	compare immediate	cc(A:imm); RP:=RP-1;
0	0	2	-	-	-	ADDS	add to S	S:=S+imm
0	0	3	-	-	-	LADI	logical add immediate	ccl(A:=A'+'imm)
0	0	4	0--	-	-	ORRI	OR right immediate	cc(A:=A|I.<8:15>)
0	0	4	4--	-	-	ORLI	OR left immediate	cc(A:=A|(I.<8:15>'<<'8))
0	0	5	-	-	-	LDLI	load left immediate	RP:=RP+1; cc(A:=imm rotate 8)
0	0	6	-	-	-	ANRI	AND right immediate	cc(A:=A&imm)
0	0	7	-	-	-	ANLI	AND left immediate	cc(A:=A&(imm rotate 8))
1	0	0	-	-	-	LDI	load immediate	RP:=RP+1; cc(A:=imm)
1	0	0xx	-	-	-	LDXI	load x immediate	cc(X:=imm)
1	0	4	-	-	-	ADDI	add immediate	ccn(A:=A+imm)
1	0	4xx	-	-	-	ADXI	add x immediate	ccn(X:=X+imm)
I	1	0	0--	-	-	BIC	branch if carry	if K then branch
I	1	1	0--	-	-	BGTR	branch if greater	if ~(N|Z) then branch
I	1	2	0--	-	-	BEQL	branch if equal	if Z then branch
I	1	3	0--	-	-	BGEQ	branch if greater or equal	if ~ N then branch
I	1	4	0--	-	-	BLSS	branch if less	if N then branch
I	1	5	0--	-	-	BNEQ	branch if not equal	if ~ Z then branch
I	1	6	0--	-	-	BLEQ	branch if less or equal	if N|Z then branch
I	1	7	0--	-	-	BNOC	branch no carry	if ~ K then branch
I	1	0	4--	-	-	BUN	branch unconditional	branch
I	1	0xx	4--	-	-	BOX	branch on X	if X<A then {X:=X+1; branch} else RP:=RP-1
I	1	4	4--	-	-	BAZ	branch on A zero	if A=0 then branch; RP:=RP-1
I	1	5	4--	-	-	BANZ	branch on A nonzero	if A<>0 then branch; RP:=RP-1
I	1	6	4--	-	-	BNOV	branch if no overflow	if ~ V then branch
I	1	7	4--	-	-	BSUB	branch to subroutine	stack[S:=S+1]:=P; branch
I	2	0xx	0--	-	-	LWP	load word from program	RP:=RP+1; cc(A:=code[branchadr+X])
I	2	0xx	4--	-	-	LBP	load byte from program	RP:=RP+1; adr:=(if indirect then code[dba] else 0) +dba'<<'1+X; A:=code[adr.<0:14> +(dba&%100000)]. <8*adr.<15>:8*adr.<15>+7>; ccb(A)
0	2	4	n	r	c	PUSH	push to stack	stack[S+1:S+c+1] :=R[(r-c)mod 8:r]; RP:=n; S:=S+c+1
1	2	4	n	r	c	POP	pop from stack	R[(r-c)mod 8:r] :=stack[S-c:S]; RP:=n; S:=S-c-1
0	2	5	0--	-	-	RSUB	return from subroutine	P:=stack[S]; S:=S-I.<8:15>

Table B-2. Instruction Definitions (Continued)

1 2 5 - - -	EXIT	procedure exit	if CSPACEID<>(stack[L-1]&
			%4437) then
			call xmap(stack[L-1]&%4437);
			S:=L-I.<8:15>;
			P:=stack[L-2];
			t:=ENV;
			ENV:={stack[L-1]&ENV&%173000}
			|{stack[L-1] & %4740}
			|{ENV & %37};
			L:=stack[L];
			if t.<0> then
			instruction breakpoint.
0 2 5 4 - -	LWXX	load word extended	cc(A:=xmem[A<<1+xbase])
0 2 6 4 - -		indexed	
0 2 5 5 - -	SWXX	store word extnded	xmem[A<<1+xbase]:=B;
0 2 6 5 - -		indexed	RP:=RP-2
0 2 5 6 - -	LBXX	load byte extended	ccb(A:=bxmem[A+xbase])
0 2 6 6 - -		indexed	
0 2 5 7 - -	SBXX	store byte extnded	bxmem[A+xbase]:=B;
0 2 6 7 - -		indexed	RP:=RP-2
1 2 5 4-- - -			*** undefined ***
0 2 6 00mssd n	MOVW	move words	while A>0 do
			{dest(C):=source(B);
			A:=A-1; B:=B+movestep;
			C:=C+movestep};
			RP:=n
0 2 6 02mssd n	COMW	compare words	N:=0; Z:=1;
			while Z and A>0 do
			{cc(dest(C)':'source(B));
			if Z then
			{A:=A-1; B:=B+movestep;
			C:=C+movestep}};
			RP:=n
1 2 6 00mssd n	MOVB	move bytes	while A>0 do
			{bytedest(C):=bytesource(B);
			A:=A-1; B:=B+movestep;
			C:=C+movestep};
			RP:=n
1 2 6 02mssd n	COMB	compare bytes	N:=0; Z:=1;
			while Z and A>0 do
			{cc(bytedest(C):
			bytesource(B));
			if Z then
			{A:=A-1; B:=B+movestep;
			C:=C+movestep}};
			RP:=n
1 2 6 40mssd n	SBW	scan bytes while	while bytesource(B)<>0 and
			bytesource(B)=A do
			B:=B+movestep
			K:=bytesource(B)=0; RP:=n
1 2 6 42mssd n	SBU	scan bytes until	while bytesource(B)<>0 and
			bytesource(B)<>A do
			B:=B+movestep
			K:=bytesource(B)=0; RP:=n

Table B-2. Instruction Definitions (Continued)

```
0 2  7  -  -  - |PCAL|procedure call     |stack[S+1:S+3]:=(P
                |    |                   |    ,(ENV & %177740)|CSPACEID
                |    |                   |    ,L);
                |    |                   |t:=I.<7:15>;
                |    |                   |if ~PRIV then
                |    |                   | {if t>=code[0] then
                |    |                   |  {if t>=code[1] then
                |    |                   |    priv trap;
                |    |                   |   PRIV:=1;
                |    |                   |  };
                |    |                   | };
                |    |                   |L:=S:=S+3;
                |    |                   |P:=code[t];
                |    |                   |RP:=7.
1 2  7  -  -  - |XCAL|external procedure |t:=(ENV&%177740)|CSPACEID;
                |    | call              |stack[S+1:S+3]:=(P,t,L);
                |    |                   |i:=SEG[CSSEG[cseg]*2].<9:15>
                |    |                   |                      *%2000-1;
                |    |                   |c:=code[i-I.<7:15>];
                |    |                   |s.<7>:=c.<0>;        ! CS
                |    |                   |s.<4>:=c.<1>;        ! LS
                |    |                   |s.<11:15>:=c.<2:6>; ! space
                |    |                   |                     !   index
                |    |                   |if s<>CSPACEID then
                |    |                   | call xmap(s);
                |    |                   |m:=2*t.<1>+t.<0>+2;
                |    |                   |t:=c.<7:15>;
                |    |                   |if ~ PRIV then
                |    |                   | {if t>=mem[m,0] then
                |    |                   |  {if t>=mem[m,1] then
                |    |                   |    priv trap;
                |    |                   |   PRIV:=1;
                |    |                   |  };
                |    |                   | };
                |    |                   |L:=S:=S+3;
                |    |                   |LS:=c.<1>;
                |    |                   |CS:=c.<0>;
                |    |                   |P:=code[t];
                |    |                   |RP:=7.
0 3  0  0  -  - |LLS |logical left shift |computeshiftcount;
                |    |                   |cc(A:=A'<<'shiftcount)
0 3  0  1  -  - |LRS |logical right      |computeshiftcount;
                |    |shift              |cc(A:=A'>>'shiftcount)
0 3  0  2  -  - |ALS |arithmetic left    |computeshiftcount;
                |    |shift              |cc(A:=A<<shiftcount)
0 3  0  3  -  - |ARS |arithmetic right   |computeshiftcount;
                |    |shift              |cc(A:=A>>shiftcount)
0 3  0 4-- -  - |    |                   | *** undefined ***
1 3  0  0  -  - |DLLS|double logical     |computeshiftcount;
                |    |left shift         |cc(BA:=BA'<<'shiftcount)
1 3  0  1  -  - |DLRS|double logical     |computeshiftcount;
                |    |right shift        |cc(BA:=BA'>>'shiftcount)
1 3  0  2  -  - |DALS|double arithmetic  |computeshiftcount;
                |    |left shift         |cc(BA:=BA<<shiftcount)
1 3  0  3  -  - |DARS|double arithmetic  |computeshiftcount;
                |    |right shift        |cc(BA:=BA>>shiftcount)
1 3  0 4-- -  - |    |                   | *** undefined ***
I 3 0xx -  -  - |LDX |load X             |cc(X:=word)
```

Table B-2. Instruction Definitions (Continued)

I	3	4xx	-	-	-	NSTO	nondestructive store	wordx:=A
I	4	0xx	-	-	-	LOAD	load	RP:=RP+1; cc(A:=wordx)
I	4	4xx	-	-	-	STOR	store	wordx:=A; RP:=RP-1
I	5	0xx	-	-	-	LDB	load byte	RP:=RP+1; ccb(A:=bytex)
I	5	4xx	-	-	-	STB	store byte	bytex:=A.<8:15>; RP:=RP-1
I	6	0xx	-	-	-	LDD	load double	RP:=RP+2; cc(BA:=dwordx)
I	6	4xx	-	-	-	STD	store double	dwordx:=BA; RP:=RP-2
I	7	0xx	-	-	-	LADR	load address	RP:=RP+1; A:=address+X
I	7	4xx	-	-	-	ADM	add to memory	ccn(wordx:=wordx+A); RP:=RP-1

APPENDIX C

HIGH-LEVEL PROCESSOR COMPARISON

This appendix provides a high-level comparison of three processors manufactured by Tandem: NonStop 1+, Nonstop II, and NonStop TXP.

Table C-1. Processor Comparison

CONFIGURATION		
NonStop 1+	NonStop II	NonStop TXP
Two-board CPU	Three-board CPU	Four-board CPU
Memory board 384K bytes	Memory board 512K bytes or 2M bytes	Memory board 2M bytes
32 I/O slots	24 I/O slots; controllers and peripherals same as NonStop 1+ except 10MB and 50MB disc are not supported; 6100 Communications Subsystem supported; 3207 Tape Controller supported	24 I/O slots; controllers and peripherals same as NonStop II

Table C-1. Processor Comparison (Continued)

CONFIGURATION (Continued)		
NonStop 1+	NonStop II	NonStop TXP
Service via Diag-Link interface	Service via OSP, PMI, and DDT interface FOX network links a maximum of fourteen systems	Service via OSP, PMI, and DDT interface FOX network links a maximum of fourteen systems
PHYSICAL MEMORY		
NonStop 1+	NonStop II	NonStop TXP
2MB physical 2MB physical address capability 500-nanosecond cycle time	8MB physical 16MB physical address capability 400-nanosecond cycle time	8MB physical 16MB physical address capability 116-nanosecond access time through cache
LOGICAL MEMORY		
NonStop 1+	NonStop II	NonStop TXP
System Code: one map, second 64K map accessed by "LIBRARYX" Four maps	System Code: one segment permanently mapped; up to 32 library segments--one mapped on call Sixteeen maps including Extended Address Cache (32 entries)	System Code: one segment permanently mapped; up to 32 library segments--one mapped on call Page Table cache (1024 entries for segments 0-15) (1024 entries for segments other than 0-15)

Table C-1. Processor Comparison (Continued)

LOGICAL MEMORY (Continued)		
NonStop 1+	NonStop II	NonStop TXP
No data cache	No data cache	Data cache (64K bytes)
16-bit address	16-bit and 32-bit address capability	16-bit and 32-bit address capability (cache accessed directly by 32-bit extended address)
Logical address limited to 512K bytes (6 segments)	Virtual address capability one gigabyte (8192 segments)	Virtual address capability one gigabyte (8192 segments)
User data 128K bytes	User data 128K bytes; multiple extended data segments, each up to 128M bytes	User data 128K bytes; multiple extended data segments, each up to 128M bytes
User code 128K bytes	User code 2 megabytes	User code 2 megabytes
No user library	User library 2 megabytes	User library 2 megabytes
System data SHORTPOOL	Process file segment; up to 128K bytes for each process	Process file segment; up to 128K bytes for each process
System data IOPOOL	Each system process manages its own data space; up to 1MB I/O buffer space	Each system process manages its own data space; up to 1MB I/O buffer space
CB space for additional system data	Segmented memory; CB space mechanism no longer required	Segmented memory; CB space mechanism no longer required

Table C-1. Processor Comparison (Continued)

I/O TRANSFER		
NonStop 1+	NonStop II	NonStop TXP
4KB maximum I/O 4KB maximum disc I/O 4MB channel transfer rate	64KB maximum I/O 4KB maximum disc I/O 5MB channel transfer rate	64KB maximum I/O 4KB maximum disc I/O 5MB channel transfer rate
INSTRUCTION MICROCODE		
NonStop 1+	NonStop II	NonStop TXP
2K words ROM	ROM 4K words RAM control store 8K words ROM entry point table 1K words	ROM bootstrap approx. 1200 words RAM control store 8K words vertical 4K words horizontal RAM entry control store 1.5K words
ERROR DETECTION		
NonStop 1+	NonStop II	NonStop TXP
Memory contents parity checked; double-bit detection, single-bit correction Map parity (software) Data paths protection: software checksum	Memory address and contents parity checked; double-bit detection, single-bit correction Map parity (hardware) Data paths protection: hardware parity	Memory address and contents parity checked; double-bit detection, single-bit correction Register parity (hardware) Data paths protection: hardware parity

INDEX

READER COMMENT CARD

Tandem welcomes your comments on the quality and usefulness of its software documentation. Does this manual serve your needs? If not, how could we improve it? Your comments will be forwarded to the writer for review and action, as appropriate.

If your answer to any of the questions below is "no," please supply detailed information, including page numbers, under Comments. Use additional sheets if necessary.

- ▶ Is this manual technically accurate? Yes ☐ No ☐
- ▶ Is information missing? Yes ☐ No ☐
- ▶ Are the organization and content clear? Yes ☐ No ☐
- ▶ Are the format and packaging convenient? Yes ☐ No ☐

Comments

Name Date

Company

Address

City/State Zip

System Description Manual

NonStop™ Systems

82507 A00

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 482 CUPERTINO, CA, U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

Tandem Computers Incorporated
Attn: Manager—Software Publications
Location 01, Department 6350
19333 Vallco Parkway
Cupertino CA 95014-9990

TAPE

TAPE

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599