HONEYWELL

MULTICS PROGRAMMER'S REFERENCE MANUAL

SOFTWARE

MULTICS

PROGRAMMER'S REFERENCE MANUAL
ADDENDUM A

SUBJECT

Changes and Additions to the Manual

SPECIAL INSTRUCTIONS

This is the first Addendum to AG91-04, dated February 1985. Change bars in the margins indicate technical changes and additions; asterisks denote deletions.

Note:

Insert this cover behind the manual cover to indicate the manual is updated with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 12.0

ORDER NUMBER

AG91-04A

January 1987

47170
5C287
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
3-1, 3-2	3-1, 3-2
3-5 through 3-8	3-5 through 3-8 3-8.1, blank
3-59 through 3-62	3-59 through 3-62
4-15, 4-16	4-15, 4-16
5-13, 5-14	5-13, 5-14
6-19, 6-20	6-19, 6-20
7-71, 7-72	7-71, 7-72 7-72.1, blank
B-7 through B-10	B-7 through B-10
B-29 through B-32	B-29 through B-32
G-1 through G-4	G-1 through G-4 G-4.1, G-4.2
G-7 through G-12	G-7 through G-12 G-12.1, blank
G-19, G-20	G-19, G-20
G-23, G-24	G-23, G-24 G-24.1, blank
	H-16.1, blank
H-17, H-18	H-17, H-18 H-18.1, blank

11/86

 File No.: 1L13, 1U13 AG91-04A

MULTICS

PROGRAMMER'S REFERENCE MANUAL

SUBJECT

Reference Material Describing the Overall Mechanics, Conventions, and Usage of the Multics System

SPECIAL INSTRUCTIONS

This publication supersedes the previous edition of the manual, Order No. AG91-03, dated February 1983, and its associated addendum, Addendum A, dated December 1983.

Section 6 is completely rewritten and does not contain change indicators. Section 10 is new and does not contain change indicators. In all other sections change bars in the margin indicate new and changed information and asterisks denote deletions. See the "Significant Changes" section in the Preface for a description of changed information.

SOFTWARE SUPPORTED

Multics Software Release 11.0

ORDER NUMBER

AG91-04

February 1985

Honeywell

PREFACE

The Multics Programmer's Reference Manual contains general information about the Multics command and programming environments. It describes such subjects as the command language, the storage system, and the input/output system.

The following are the primary reference manuals for user and system programmers of the Multics system. These manuals contain general information and may be referenced throughout this document. For convenience, these references are as follows:

Document	*Referenced In Text As*
Multics Programmer's Reference Manual (Order No. AG91)	Programmer's Reference
Multics Commands and Active Functions (Order No. AG92)	Commands
Multics Subroutines and I/O Modules (Order No. AG93)	Subroutines

Each section/appendix of this document is structured according to the heading hierarchy shown below. Each heading indicates the relative level of the text that follows it.

LEVEL	HEADING FORMAT
1 (highest)	**ALL CAPITAL LETTERS, BOLD TYPE FACE**
2	**Initial Capital Letters, Bold Type Face**
3	*ALL CAPITAL LETTERS, ITALICS TYPE FACE*
4	*Initial Capital Letters, Italics Type Face*

 File No.: 1L13, 1U13 AG91-04

Significant Changes in AG93-04

The description of the Multics storage system in Section 2 has been updated to reflect the support of extended entry types.

Section 3 contains new material on date/time values.

Section 4 now includes information describing the inplementation of extended entry types.

The description of the Multics input/output system in Section 5 has been updated to reflect the addition of a new I/O module (mtape_) used for tape input/output. Since the new mtape_ I/O module accepts a file open, close, and detach description, the programming/implementation instructions have also been updated to reflect this capability.

Section 6 (Multics Security) has been completely rewritten.

New conditions generated by the data management software have been added to Section 7.

Section 10 is a new section describing the new data management software.

There are changes to the description of the Multics standard magnetic tape format in Appendix F.

CONTENTS

Illustrations

Tables

SECTION 1

MULTICS CONCEPTS AND CHARACTERISTICS

The first part of this section is a brief introduction to the Multics system. Many items mentioned here are described in detail in other sections of this manual. Refer to the contents and index of this document to locate desired information. When necessary, the user is referred to other manuals.

The second part of this section is a glossary of Multics terms. A reference that directs the reader either to a section of this manual or to another manual is supplied with most of the terms defined.

SYSTEM CONCEPTS

Multics is a unique combination of hardware, software, communications capabilities, and supervisory techniques. The system provides capabilities that have long been sought by research, government, academic, and network-oriented computer users--those users who require unique security, system development, and centralized data base features.

Setting Multics apart from other offerings in the general purpose computer market is its ability to provide total resources on demand. Computer systems previously have been measured in terms of memory size, speed, and hardware cost, but Multics is gauged by its ability to provide the most cost-efficient environment for problem solving. Productivity of the system is high because all Multics software--including the operating system supervisor, user programs, and data files--is free of main memory constraints and of any particular hardware configuration.

The ability to share data within the framework of a general purpose, time-sharing system, is a vital feature of Multics that can be directly applied to administrative problems, applications requiring a multiuser accessible data base, and general application of the computer to very complicated problems. The attention paid to mechanisms to provide and control privacy is of direct interest for many applications dealing with proprietary information.

Multics offers a number of additional capabilities that go well beyond those provided by many other systems. Those which are most significant from a user's point of view are described in this section. Perhaps the most important aspect of all is that a single system comprises all of these capabilities simultaneously. The major design concepts of the Multics system include:

- Virtual memory designed to make addressable memory seem virtually infinite
- Selective sharing of information through controlled access that is regulated by both software and hardware
- Security mechanisms enforced by hardware: this includes the Multics ring structure
- Structural administration, allowing decentralized control and management of system resources
- Flexible user interfaces, allowing a wide variety of programming environments
- Remote terminals as the normal mode of system access
- Efficient service to all users whether their use of system resources is very large or very small
- Continuous operation through the use of dynamic hardware configuration techniques and online software maintenance and system administration
- Open-ended, modular system design that anticipates the evolution of technological improvements and the expansion of user requirements

SYSTEM CHARACTERISTICS

The following paragraphs describe the major characteristics of the Multics system. These characteristics are integral parts of the Multics system and cannot be separated from the system--yet in many instances, use of these capabilities is optional to the individual user.

Segments

The segment is the unit of storage of the Multics storage system analogous to a file on other systems. A segment can range in size from 0 to 255K (K equals 1024) 36-bit words. On Multics, all information is grouped into nondirectory and directory segments. A nondirectory segment is a collection of instructions or data specified by a user. A directory segment is a catalog of subordinate segments, created and maintained by users via the supervisor. The directory concept is the key to several Multics features including storage structure, administrative control, search rules, and naming conventions.

A user can create a segment by issuing a command (e.g., create) from command level or via a call statement from within a program. A user has control over every segment he creates. The segment attributes mentioned above provide the user with extensive control over the manipulation and sharing of the segments he creates. (See "Selective Sharing" below.) A user may specify the individuals who have access to his segments. Also levels of protection (rings) can be specified as a further control over the same segments.

Virtual Memory

The Multics virtual memory makes all segments in the storage system directly addressable. An address in Multics, as used by the hardware, consists of two components: the first identifies a segment and is called a segment number; the other specifies a location within that segment.

A segment number is assigned by the supervisor and associated with the specified segment by user request, provided the user has the necessary access privileges. This request is often done implicitly as part of some other supervisor function.

Once a segment number has been provided by the supervisor, user software can reference the segment directly with appropriate machine instructions. The data or code of the segment so referenced is automatically brought into main memory, if necessary, so that the processor can use it.

Since the physical movement of information between secondary storage and main memory is totally automatic, it is of no concern to the programmer when he is constructing his application. A user does not have to be concerned with where and on what devices his segments reside. Because of the demand paging technique described below, users need not be concerned about overlaying or partitioning program modules to satisfy limited main memory resources. Since conventional input/output is not required programming on Multics is greatly simplified.

Paging

Since segments can be different sizes, it may be impractical to have an entire segment in main memory when in use. Therefore Multics segments are automatically subdivided into fixed-size (1024 words) storage units called pages. When a segment is referenced, the page referenced is automatically retrieved from secondary storage and placed in any available "frame" in main memory. When main memory is filled and more frames are needed, some pages have to be displaced. Pages not used recently are moved to secondary storage so that new pages may be transferred to main memory.

Address mapping at the hardware level allows the system to determine whether or not a page of a referenced segment is in main memory. If the page is not in main memory, a missing page exception occurs (called a "page fault"). The system software intervenes at this point and processes the page fault by locating the desired page of the storage system in secondary storage and transferring it to main memory. This procedure is automatic; and the time involved is not noticeable to the user. During this phase, the process that generated the page fault may relinquish control of the processor and the system may dispatch the processor to another process. (See "Process" below.) Once the page does arrive in main memory, the system notifies the "waiting" process and schedules it for continued execution. Consequently, only those pages that are currently needed are in main memory at any one time.

Process

A process may be defined as the way the system is seen by a logged-in user; in essence, a user's process is the user's (virtual) machine. Multiprogramming multiplexes real processors between users' processes, making it appear that each user has a processor at his service. A user's process executes programs sequentially.

The system creates a process for each user at login time. (For information on logging in and out on the system, see the *New Users' Introduction to Multics, Part I*, Order No. CH24.) Within the constraints imposed by the supervisor, users and project administrators may customize their processes as desired: the commands, command processor, and environment provided by the system can all be replaced in users' processes by their own code.

Each user's process executes programs fairly independent of other users' processes. Information may be shared between processes, allowing sharing of programs and communication between processes (if desired).

Each process has its own address space and is made up of three parts. These parts are described in the process' descriptor segment; basically, they are the process' characteristics in ring 0, ring 1, and ring 4, the user ring (for a further discussion of Multics ring structure, see "Selective Sharing" below). For instance, each process has a separate stack and linkage area for each ring that it executes in. On the other hand, each process has other attributes that are the same no matter which ring it is executing in. A good example of that is the descriptor segment. Figure 1-1 indicates which things are per process and which per ring.

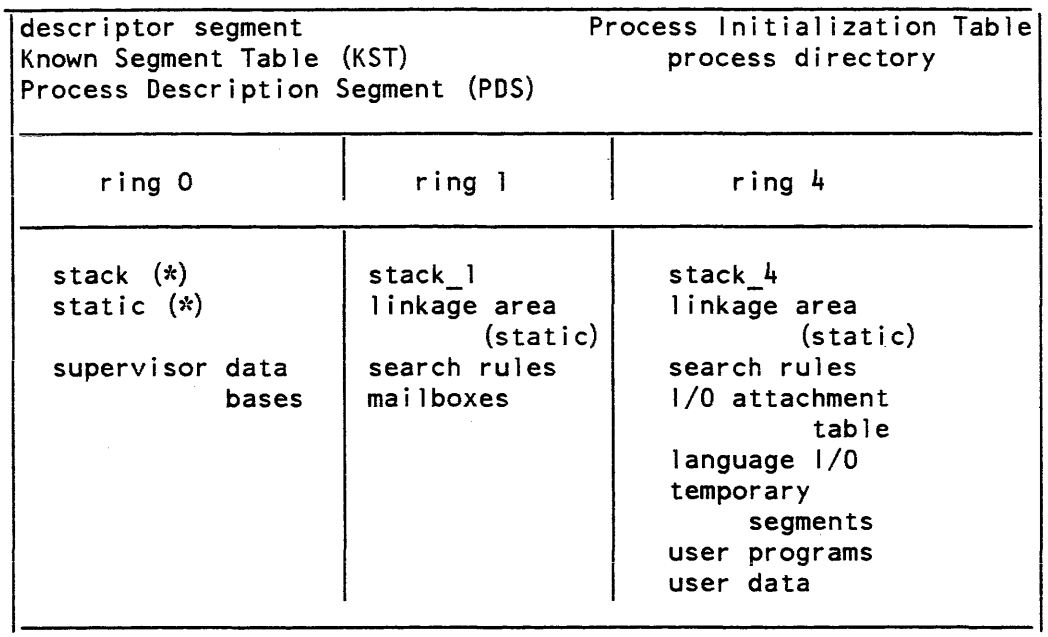


Figure 1-1. Process Characteristics Per Ring

Ring 0 comprises the segments of the central supervisor. It is in this ring that each process' segment numbers are assigned. The Known Segment Table, in which these numbers are kept, then applies, of course, to an entire process, and that process can directly reference only those segments that have been assigned a segment number by calls to supervisor programs.

Ring 1 contains system routines such as those that manage mailboxes and queues. Ring 2 is used, in large part, for system data bases. Ring 3 is reserved mostly for other administrative routines such as those that are specific to a particular site. Most user processes start running in ring 4. Rings 5 through 7 are used by programmers writing their own protective subsystems. The classic example of this is a teacher who runs student programs in ring 5 and the grading program in ring 4. Rings 6 and 7 lack access to most system software, so the subsystem designer can create severely limited
programming environments.

SELECTIVE SHARING

Segments are data objects that exist independently of any process. The system manages the physical location of pages of segments. If the pages are in use, they will be in main memory. If several users have the same segment known in their process, they will reference the same physical locations of main memory when referencing it. Per-user "copies" or "images" of segments can be created under special circumstances, but generally segments are shared. Hence, several users referencing a given segment may use its contents to communicate, given that access has been appropriately granted. Furthermore, several processes using the same program use the same physical pages, contributing to effective use of main memory.

Access Control List

Each segment has an access control list (ACL) that names the individuals who have access to the segment and describes the type of access they have. Through the ACL, a user can grant specific access to individual users or groups of users to permit easy, controlled sharing of information. Different access rights can be granted to different users of the same segment. The hardware enforces access control during the execution of each individual machine instruction.

Access Isolation Mechanism

The access isolation mechanism (AIM) allows administrators of the system to define several levels of privilege, which the system itself rigidly enforces. Enforcing the separation of these levels is totally independent of other access controls or user action. The use of this administrative mechanism ensures privacy by preventing inadvertent disclosure of information between these privilege levels, even by those who own the information.

Ring Structure

A further refinement to selective sharing is provided by special hardware that implements the Multics ring structure. Ring structure is an advanced form of protection capability that permits the ready construction of protected data bases. Privileged users may have complete access to the data base and can control (by program) the information less privileged users can see.

Logically, the ring structure is eight concentric rings, each representing the level of privilege accorded to procedure segments executed in that ring. The highest level of privilege is the innermost ring, designated as ring 0; the outermost is ring 7. Privileged ring segments, such as a supervisor and special user subsystems, are protected from uncontrolled use by less-privileged rings. These segments can only be used by procedures in less-privileged rings if called via a special "gate" mechanism, and the user must have access to the individual segment as well.

The Multics ring implementation makes it possible to:

- Create protected programs and data bases for controlled use by other users
- Implement a supervisor program in rings with differing degrees of privilege
- Debug a program in an unprivileged ring and then move it to a privileged ring with no recompilation or modification

SYSTEM ADMINISTRATION

All information stored online on Multics is organized into a tree-structured hierarchy. Within this hierarchy, directories catalog segments residing below them in the tree. (See Section 2, "Storage System.") The organization of the body of users on Multics is patterned after the organization of the storage system. Users are grouped by the system administrator into projects, which are generally under the control of a project administrator. The project administrator may impose special disciplines on users within his project. For example, the project administrator defines the initial procedure in a process for users under his project. The project administrator also allocates storage quota to individual users based on the quota granted his project by the system administrator. The distribution of authority between the system administrator and project administrator results in a decentralized control of the system. *

The facilities required to manage a Multics site are integrated into the system itself. In the area of financial control, the Multics system accounts for use of resources on a per-user basis and organizes these accounts based upon project and system administration. Users can be allocated quotas according to storage space, central processor utilization, or dollar amounts based on the current billing rates. Users, project administrators, and the system administrator can interrogate quota amounts and usage at any time.

USER INTERFACES

Multics has an open-ended design with a uniform interface for both user-written and system-provided commands. The user can create or manipulate segments residing in various user directories while at command level or from within a program. Users can create commands and subsystems of arbitrary complexity. The interfaces available to system-provided commands and subsystems are available to the user and are documented elsewhere in this manual (see in particular, Sections 4 and 5, and Appendixes G, H, and I).

ENVIRONMENT SHAPING

A Multics user is not restricted to the programming environment defined by the standard system but can alter this environment for private use or use an altered environment that a project administrator provides or imposes on him. As an example, the project administrator may offer some of his users only a subset of the full system (a limited service system), or he may create a completely separate environment (a closed subsystem) that bears no resemblance to the standard Multics environment and requires no knowledge of the Multics system itself. These environment changes are made possible by a large number of Multics mechanisms. Primary contributors are:

- Modular system design that allows easy replacement of a specific operating system module without affecting other modules
- Implementation of the system in the PL/I language, which permits easy interfacing with operating system modules

- Project administration features, which permit the installation and management of a new environment
- Security and protection features, which keep the environment separate from the other users in an atmosphere of mutual protection

SYSTEM SOFTWARE

The Multics system includes a full complement of software facilities. These software facilities include several language processors, various communications products, a data base management capability, word processing software, and a host of specialized facilities for various utility functions. These software facilities are listed below.

PL/I

The PL/I compiler for Multics offers a full selection of language facilities and access to the advanced features of Multics. PL/I is the recommended programming language for Multics users. Multics PL/I conforms to the American National Standards Institute standard (ANSI X3.53-1976) for the language and the ISO International Standard 6160-1979.

FORTRAN

The Multics FORTRAN compiler conforms to the American National Standards Institute (ANSI X3.9-1978) FORTRAN. Multics FORTRAN is a superset of the ANSI standard FORTRAN and is source language compatible with L66/GCOS8 FORTRAN. Multics FORTRAN complies with the Federal Information Processing Standard (ANSI X3.9-1978) for the FORTRAN language.

BASIC

The Multics BASIC is compatible with the Dartmouth Version 6 BASIC and contains all the functional capabilities of the L66/GCOS8 BASIC compiler. Except for minor differences, it conforms to the ANSI Standard for Minimal BASIC (ANSI X3.60-1978).

COBOL

This compiler is a subset of the ANSI standard COBOL (ANSI X3.23-1974) and of the Federal Information Processing Standard (FIPS 21-1). It is source language compatible with L66/GCOS8, COBOL-74.

APL

APL is a powerful interpretive language available to Multics users. The Multics APL interpreter is compatible with other common APL implementations.

PASCAL

Multics PASCAL is based on the standard ISO Pascal. Extensions are available to augment the standard and make PASCAL programming on Multics easier and more versatile. Pascal is a popular language because of its carefully chosen control structures and powerful data structuring capabilities.

ALM

ALM is the assembly language on Multics. It is commonly used for privileged supervisor code, compiler support operations and utility packages. It is not recommended for general use.

QEDX

The qedx editor is used to create and edit ASCII segments. Through its macro capabilities, it also qualifies as a minor interpretive language.

TED

The ted editor is used to create and edit ASCII segments. It is an extended version of qedx, designed to facilitate macro writing.

EMACS

The Emacs editor is an integrated editing, text preparation, and screen management system designed to take advantage of the features of modern display terminals.

Communications Software

Multics supports a wide variety of specialized tools for communications support. Both synchronous and asynchronous protocols are supported. Multics can be configured as a host system for remote job entry workstations or, conversely, Multics can be used as a remote workstation to a foreign host. Also supported is a capability to allow DPS 6 systems to be used as satellite systems via an X.25 protocol interface facility.

Sort/Merge

The Sort/Merge subsystem provides generalized file sorting and merging capabilities, specialized for execution by user supplied parameters. The Sort orders an unranked file according to the values of one or more specified key fields in the user's records. The Merge collates the contents of up to ten ordered files according to the value of one or more key fields. Input and output files associated with the Sort/Merge subsystem can have any file organization and be on any storage medium. Records can be either fixed or variable length.

GCOS Environment Simulator

The GCOS environment simulator, together with several Multics facilities, permits GCOS batch-processing jobs to be run under the control of Multics and provides some job-scheduling facilities. Invoked via the Multics gcos command, the simulator immediately runs one GCOS job in the user's process. The user's terminal is treated as the GCOS operator's console.

Multics Graphics System

The Multics Graphics System (MGS) provides a general purpose terminal-independent interface through which user or application programs can create, edit, store, display, and animate graphic constructs.

Multics Data Base Manager

The Multics Data Base Manager is written to interface with any programming language that supports a call statement. It is based on the Multics relational data store (MRDS). MRDS supports the relational model of data base organization.

Multics Report Program Generator

The Multics Report Program Generator (MRPG) is a language translator used to generate a PL/I source program from an MRPG source program with the purpose of generating formatted reports.

Logical Inquiry and Update System

The Logical Inquiry and Update System (LINUS) is a facility for accessing centralized MRDS data bases, as well as user-defined private data bases. The complete data base management capability provided by LINUS includes both retrieval and update operations. The LINUS language (LILA) is used to specify data; it is a high-level nonprocedural language that can be understood by individuals who are not necessarily computer specialists. The LINUS report generator permits both technical and non-technical users to obtain formatted reports.

Word Processing

The Multics word processing system, WORDPRO, consists of a set of commands that assist users in the input, update, and maintenance of high quality documents. The commands provide tools for text formatting, Speedtype, dictionaries for hyphenation and spelling, and list processing.

Extended Mail Facility

The Multics Mail Facility consists of the read_mail, send_mail, and print_mail commands, which enable users to send, receive, and process mail.

Executive Mail

The Multics Executive Mail Facility uses menus to assist users in sending, reading, and processing mail.

Forum

Multics Forum is an online meeting system that enables users to attend a meeting by reading proceedings that have been stored online and entering proceedings for other participants in the meeting to read.

Executive Forum

The Multics Executive Forum facility is a menu-driven interface to the Forum subsystem. Users are provided with a list of operations that can be selected by typing a number or letter.

Transaction Processing Tools

The Multics Transaction Processing subsystem provides a specialized environment for applications that interact with a data base.

The FAST/DFAST Facility

FAST and DFAST operate as subsystems under Multics to provide a time-sharing environment supporting BASIC and FORTRAN program development. (The DFAST command repertoire and language conventions are based on the Dartmouth Time-sharing System with extensions for compatibility with Multics.)

Menu Creation Facilities

Multics menu creation facilities consist of commands and subroutines that can be used by application programmers to create and manage menus.

Inter-Multics File Transfer Facility

The Inter-Multics File Transfer Facility permits users to easily transfer files and subtrees between Multics systems.

Report Writer

The Multics Report Writer (MRW) provides the capabilities to utilize formatted data extracted from sources other than Multics Relational Data Store data bases. It consists of an end-user oriented subsystem request interface, and a programmer subroutine interface. It was designed to serve the needs of the casual and experienced user.

File Transfer To and From Personal Computers

Multics supports three protocols that enable users to transfer files between Multics and microcomputers. These protocols consist of a file transfer protocol, Kermit, and two data transfer protocols, XMODEM and IBM PC-to-Host (Multics). Kermit consists of the basic Kermit file transfer protocol and the Kermit server. XMODEM and IBM PC-to-Host (Multics)are implemented by IO modules that may be used to transfer files by way of either the micro_transfer command or through a user-created file interface system. Users wishing to design their own data transfer protocols may also use the micro_transfer command to act as an interface between the Multics file system and the respective user data transfer protocol.

Other Support Facilities and Tools

The probe, debug, and trace commands permit a user to analyze and correct a compiled program at both the original source level and the more specific machine register level.

The Multics Subsystem Utilities (ssu_) provide a general-purpose interface for implementing interactive subsystems, such as the Extended Mail Facility and Forum, in an environment analogous to Multics command level. Detailed information on the use of this facility is contained in Section 4.

Performance measurement tools permit the user to analyze his program's behavior so that optimum applications software can readily be developed.

Interuser communication facilities, both immediate and deferred, permit online messages to be transmitted among users.

Online documentation facilities provide the user with useful information and document preparation tools.

For easy reference, the standard commands and subroutines provided by the Multics system are listed according to function in the respective Commands and Subroutines manuals.

ACCESS TO THE SYSTEM

The primary means of accessing the Multics system is from a remote terminal. The system accepts input from a terminal, interprets the user's request, and invokes the software component to perform the desired function. The software component can be either system or user supplied: there is no distinction at the command or subroutine level. The command language allows recursive, iterative commands and the embedding of function calls in the command line.

The command processor is a shared, replaceable module, written in PL/I. The design of the command processor thus permits an extremely wide range of interfaces to all system facilities either on a controlled or open-ended basis. The project administrator can require a user to interface with a special version of the command processor, thereby limiting the software requests or commands available to him.

The Multics system does not usually differentiate between interactive and batch users, except that a batch user (called absentee in Multics) is not available to answer any questions the system may ask and must therefore anticipate such questions and have prepared answers ready.

SERVICE TO LARGE AND SMALL USERS

The Multics system automatically assigns system resources to a user in proportion to the size of his task. System functions (such as locating and retrieving information from secondary storage) are invoked on a demand basis, as the detailed requirement is encountered by the program. This not only relieves the programmer of the burden of predicting in advance his use of system resources, but also prevents an additional burden on the system caused by programmers calling for more resources than they need. By default, the system controls the automatic allocation of resources for all users. In addition, the system and project administrators can set storage quotas on a user and even impose limits on the amount of system resources that the user can consume in a given time interval.

SYSTEM DESIGN

The designers of the Multics system were determined from the very beginning to develop a system that could both evolve as a body of software on a given machine and sustain a movement across hardware generations. To attain this goal, they implemented a modular design. Operating system modules may be easily replaced on a system or individual user basis. In addition, most of the Multics operating system is written in PL/I, which makes the system even more flexible and easy to modify.

Continuous Operation

Various system features contribute to the Multics characteristic of continuous operation:

- Central processors and memory units may be added or subtracted without shutting down the system
- User programs and the system itself need not change structure in any way whatever due to differences in hardware configuration
- Tasks required to manage the system can be performed without interrupting service; these tasks include metering system or user behavior, invoking management subsystems such as accounting and billing, or even updating the bulk of the system software capabilities and facilities

System Reliability

Information stored online on a Multics system is protected by two distinct types of backup systems, one which uses the normal hierarchy mechanisms to access data, referred to as the hierarchy backup system, and another which operates at the physical storage volume level, referred to as the volume backup system. Both backup systems dump onto magnetic tape any segment whose contents have been changed during the backup interval. The length of the backup interval and the segments to be protected can be set by the system administrator. A straightforward technique permits the retrieval of a segment from either type of backup tape and its reinclusion in the online storage system. There also is a subsystem called the "salvager" that examines the online storage system after a failure, corrects improper directories, and informs operations personnel of missing or damaged segments. These may then be retrieved from the backup tapes. Another subsystem, the "scavenger," performs the same function as the physical volume salvager without having to remove the affected volume from service.

GLOSSARY OF MULTICS TERMS

absentee
: A facility for running background jobs (noninteractive processes). (See the enter_abs_request command in the Commands manual.)

access attributes
: See access modes below.

access class
: An access isolation mechanism (AIM) attribute that denotes the sensitivity of information contained in a segment, directory, multisegment file, or message in a message segment. An access class is associated with an entry for its lifetime. (See "Nondiscretionary Access Control" in Section 6.)

access control
: The mechanism for determining who can reference or modify segments (files) and directories. (See "Discretionary Access Control" in Section 6.)

access control list (ACL)
: A set of access identifiers specifying who can access a segment or directory. Associated with each access identifier is a set of allowed modes of access to that segment or directory. There is an ACL for each segment and each directory. See initial access control list below. (See "Discretionary Access Control" in Section 6.)

access isolation mechanism (AIM)
: The mechanism used to guarantee that only authorized persons access certain classes of information. (See "Nondiscretionary Access Control" in Section 6.)

access modes
: A way to identify the kinds of access that can be set for a segment or directory. The access modes for segments are read (r), write (w), execute (e), and null (n). Those for directories are status (s), modify (m), append (a), and null (n). See extended access below. (See "Discretionary Access Control" in Section 6.)

active function
A function specified in a command line whose value (a character string) becomes part of an expanded reevaluated command line. (See "Active Strings" in Section 3.)

AIM
See access isolation mechanism above.

ALM
The assembly language on Multics, used primarily for programs that must closely interface with the hardware. (See the alm command in the Commands manual).

alternate name(s)
A segment, directory, multisegment file, or link can have more than one name and may be referred to equally well by any one of its names. One of the names is the primary name. The others are called secondary names or addnames. A segment often has more than one name because it is a program with alternate entry points; commands often have short names as well as long ones for convenience in typing (i.e., cwd instead of change_wdir). (See primary names below.)

answering service
The subsystem that runs in the Initializer process and logs users in and out. (See Initializer.)

archive
A segment used to conserve space. When storing a group of segments, the contents of the individual segments can be packed together in an archive to eliminate breakage in the last page of each segment. (See the archive command in the Commands manual.)

attach
The act of associating an I/O switch with a file, or other I/O switch. For example, the normal output switch (user_output) is usually attached to the terminal, but can be attached to a file via the file_output command. (See I/O module below.)

authorization
An access isolation mechanism (AIM) attribute of a process that denotes the range of information the process can access. An authorization is associated with a process for its lifetime. (See Section 6.)

backup
See hierarchy backup and volume backup below.

before journal
A storage unit that is used for storing the images of a data management file before it is modified. Before journals are implemented as extended entry types.

bind
See bound segment below.

bit count
An index to the last bit of useful information in a segment. For example, a

segment that contains 43 characters starting at the beginning has a bit count of 387 (9*43). (A segment may, however, contain useful data independent of its bit count.) (See "Entry Attributes" in Section 2.)

blocked
: The state a process is in when it is not executing and is waiting for some event to occur (such as the user typing a command line).

bound segment
: A group of (usually related) object segments bound into one object segment to save space and speed up references (calls, etc.) between them. The process of binding segments is similar to linkage editing on other systems and is done by use of the bind command. (See the bind command in the Commands manual.)

branch
: An item cataloged in the storage system that is not a link.

canonicalization
: The conversion of a terminal input line into a standard (canonical) form. This is done so that lines that appear the same on the printed page, but that may have been typed differently (i.e., characters overstruck in a different order), appear the same to the system.

carriage return
: A carriage return means that the typing mechanism moves to the first column of the next line. On the Multics system, this action is the result of the ASCII line feed character. The terminal type determines which keys the user presses to perform the equivalent action (e.g., RETURN, LINE SPACE, or NL).

character
: A hardware-related unit of information that on the Multics system is 9 bits or 6 bits. The Multics system native mode character set is 9-bit ASCII, although the hardware does support additional character sets.

closed subsystem
: A separate environment that bears no resemblance to and has no knowledge of the Multics system itself. (See "Programming Environment" in Section 1.)

command
: A program designed to be called by typing its name at a terminal. Most commands are system maintained, but any user program that takes only character string input arguments and no output arguments can be used as a command. (See "Command Language" in Section 3.)

command level
: The process state in which lines input from a user's terminal are interpreted by the system as a command (i.e., the line is sent to the command processor). A user is at command level at login when a command completes or encounters an error, or is stopped by issuing the quit signal. Command level is normally indicated by a ready message. (See "Command Environment" in Section 3.)

command processor
: The program that interprets the lines input at command level and calls the appropriate programs, after processing parentheses and active functions. (See "Command Environment" in Section 3.)

component (of an archive)
One of the segments placed in an archive. (See the archive command in the Commands manual.)

component (of an entryname)
A logical part of an entryname. Entryname components are separated by a period. (See suffix below and "Entrynames" in Section 3.)

control argument
An argument to a command that specifies the command's execution in some way. System control arguments begin with a hyphen, such as -all, -long, or -hold. The meaning of each control argument accepted by a specific command is given as part of the description of the command. Many control arguments have standard abbreviations such as -lg for -long.

crash (FNP)
an unplanned termination of service from the front end network processor causing a disconnection of the process. The process can be saved and reconnected when the -save_on_disconnect control argument has been used with the login command.

crash (system)
An unplanned termination of system availability caused by problems in hardware and/or software, often signalled by the message: MULTICS NOT IN OPERATION. Processes cannot be reconnected after a system crash.

data management file
a unit of storage within the Multics storage system. Data management (DM) files can be implemented with concurrency control and recovery support. At present, the ability to use DM files is available only to programs accessing files through the Multics Relational Data Store (MRDS) facility.

daemon
One of several system service processes that perform such tasks as process creation, backup, network control, and printing segments on the line printer.

detach
Inverse of attach (see above).

directory
A catalog of entries in the storage system. The directory contains information about the attributes of these entries and information about the physical device on which the data is stored. (See Section 2, "Storage System.")

directory (home)
The directory that is the working directory of a user when he first logs in to the system (also known as the initial working directory). Usually this directory has a pathname of the form:

```
>udd>Project_id>Person_id
```

See directory (working) below.

directory (master)
A directory whose segments reside on a different logical volume than those of its parent directory.

directory (parent)
The immediately superior directory.

directory (upgraded)
Any directory that has an access class greater than that of its parent.

directory (working)
Identifies the user's current location within the storage system with regard to pathnames. Any pathname the user types that does not begin with a greater than (>) character is considered relative to the working directory. By default, this directory is used by the search rules. (See "Search Rules" in Section 4.)

directory hierarchy
The tree-structured organization of the logical contents of the Multics storage system. (See Section 2, "Storage System.")

discretionary access control
Allows individual users to grant or deny other users access to their segments and directories at their own discretion.

dprint, dpunch (for Daemon print and Daemon punch)
A queued request to the system to output on a line printer (or card punch) the contents of a segment or multisegment file. (See the enter_output_request, dprint, and dpunch commands in the Commands manual.)

dump
See hierarchy backup below.

dynamic linking
The resolution of symbolic external references at execution time (that is, the first time the symbol is actually referenced). (See link pair below and "Dynamic Linking" in Section 4.)

effective access
The actual access mode that the system enforces for each reference or use of a segment or directory. Nondiscretionary access control can restrict, but not extend, the access granted by the discretionary controls.

entry
An item cataloged in the Multics Storage System (e.g., segments).

entry bound
For protection purposes, control must not be passed to a gate procedure at other than its defined entry points. To enforce this restriction, the first n words of a gate segment with n entry points must be an entry point transfer vector. To ensure that only these entries can be used, the hardware enforced entry bound of the gate segment must be set so that the segment can be entered only at the first n locations. See the hcs_$set_entry_bound description in the Subroutines manual.

entry point
An address in an object segment referred to by a symbolic name; e.g., that which would be produced by the PL/I or FORTRAN procedure, subroutine, or entry statements.

entry point name
The name associated with an entry point in an object segment. The entry point name is found by the dynamic linker. (See "Entry Point Names" in Section 3.)

entryname
A name given to an item cataloged in the storage system. It may contain one or more components, separated by periods. All names given to entries within one directory are unique, but need not be different from names defined in other directories. (See "Entrynames" in Section 3.)

equal convention
A method used by many commands to specify one or more characters in a group of entrynames. (See "Equal Convention" in Section 3.)

error codes
See status codes below.

exclamation point convention
See unique name below.

exec_com (ec)
A facility for executing a list of commands taken from a segment. It includes argument passing and conditional branching capabilities. (See the exec_com command in the *Multics Commands and Active Functions* manual, Order No. AG92.)

extended access
An additional field of access modes used with certain extended entry types to further restrict operations on the entry type. (See "Access Modes" in Section 5.)

extended entries
Storage system entries that represent an extension beyond the five standard entry types (segments, directories, links, MSFs, and DM files) are called extended entry types. These extended entries are used to implement special functions, usually by using ring brackets to protect the data contained therein. Most file system commands have been enhanced to operate on all types of extended entries, performing operations as if the extended entries were segments. This is done using a standard subroutine interface, fs_util_, for performing the operations, rather than calling the hcs_ subroutine directly. Each of the extended entry types is identified by a suffix appended to the entryname. The system-supplied extended entry types are listed below, but a programmer can create others:

Name	Suffix
mailbox	.mbx
forum meeting	.forum
message segment	.ms
before journal	.bj
person name table	.pnt

fault
: A hardware signal similar to an interrupt that may cause the signalling of a condition. (See "Faults" in Section 7.)

file
: A term that stands for segment and/or multisegment file.

frame
: See stack below.

gate
: The only point at which a procedure in an outer ring can transfer to a procedure in an inner ring. (See "Intraprocess Access Control" in Section 6.)

hardcore (hardcore supervisor)
: The set of routines that perform the supervisory functions of the system. The hardcore executes in ring 0.

help files
: See info segments below.

hierarchy backup
: The hierarchy backup system dumps (copies) user segments and directories onto removable storage (magnetic tape). The dumping is conventionally done using the processes Backup.SysDaemon and Dumper.SysDaemon. The information dumped can be recovered by the operations staff at the user's request. (See Section 8, "Backup.")

home directory
: See directory (home) above.

impure procedure
: A procedure that modifies itself. Such a procedure is not recommended.

info segments
: The segments whose contents are printed by invoking the help command. These segments, sometimes called help files, give information about the system. The system info segments are kept in the directory >documentation>info_segment. (>doc>info). The info segments that are peculiar to an installation are kept in >doc>iml_info_segments. (See the help command in the Commands manual.)

initial access control list
: A list that specifies what the access control list of a newly created segment or directory will be. There are separate initial access control lists for segments and directories for each ring. (See "Initial ACLs" in Section 6.)

initial working directory
See directory (home) above.

Initializer
The system control process that logs users in and out and keeps accounting statistics. This is the only process that creates and destroys other processes. Its access identifier is Initializer.SysDaemon.z.

initiate
The act of associating a reference name with a given segment in the storage system. The segment must be part of the user's "address space" (made known), and the supervisor entries will do this automatically if necessary. A reference name is said to be initiated for a given segment. (See "Reference Names" in Section 3.)

I/O module
A program that processes input and output requests directed to a given switch. It may perform operations on other switches, or call the supervisor.

I/O switch
See switch below.

IO.SysDaemon
The User_id of the system process that does dprinting and dpunching.

IOSIM
Obsolete term. See I/O module above.

library_dir_dir (ldd)
The starting directory of the subtree in which the source and object modules of the system are stored. (See Section 2, "Storage System.")

limited service system
A subset of the Multics system imposed on users by the project administrator. (See "Programming Environment" in Section 1.)

link
(1) An entry in a directory that specifies the pathname of an entry in another directory. It allows references to items in other directories as if they were actually contained in the working directory. Links eliminate the need for multiple copies of segments.
(2) An external symbolic reference. See link pair below.

link pair
An indirect word in a procedure segment's linkage section through which all references to some external data or procedure are made. Until the link is snapped, it contains symbolic information about the external object. A link pair initially contains a code that causes a fault, and invokes the dynamic linker, when first used in a process. The linking, if successful, puts the actual address of the procedure or data referenced in the link pair.

linkage section
(1) The portion of a procedure object segment that is a pure template for impure data needed by the procedure at runtime.
(2) The impure copy made from this template. (See dynamic linking above.)

listener
The program that reads command lines from the terminal and passes them to the command processor.

logical volume
A set of physical volumes that are always mounted together.

mailbox
See Person_id.mbx.

main memory frame
A 1024 36-bit word block of main memory that holds a page of a segment. (See "Paging" in Section 1.)

making a segment known
Specifying the pathname of a segment to the supervisor, and receiving a segment number in return. The segment may then be referenced by that segment number in the process. (See "Making a Segment Known" in Section 4.)

master directory
See directory (master) above.

memory units
A measure of the usage a user makes of the system memory resources.

message segment
A special type of segment that is managed by Multics supervisor programs and is not directly accessible to the user. A message segment is simply a permanent place to hold interprocess messages, e.g., dprint and dpunch requests.

Multics card code (MCC)
A code for punched card input and output. It is essentially the IBM standard EBCDIC card code. This is the default code for the dpunch command. (See "Punched Card Codes" in Appendix C.)

multiple names
See alternate names above.

multisegment file (MSF)
A file that occupies more than one segment, i.e., a file larger than 261,120 words. May only be manipulated by certain programs. (See "Multisegment Files" in Section 2.)

nondiscretionary access control
Also referred to as administrative access control, it is used to restrict discretionary controls in order to enforce the policies of the system administrator and of the organizations served by the system. The system administrator (through AIM) guarantees that only authorized persons may access certain classes of information.

object segment
A procedure or data segment produced as the result of a compilation with a system-defined format. An executable object segment can be directly executed by a process. Object segments may also be searched and linked to by the dynamic linking mechanism. (See "Creating an Object Segment" in Section 4.)

page
A 1024 36-bit word block of data within a segment.

page control
The routines that manage the transfer of pages between secondary storage and main memory frames. (See "Paging" in Section 1.)

parent directory
See directory (parent) above.

password
A character string that enables an individual user to enter the system; it is known only to that user and the program that controls access to the system. When supplied with the user's Person_id at log in time, it validates the true identity of the user. A password can be from one to eight characters long. The characters may be any characters from the ASCII character set except space and semicolon. The backspace character is also allowed and is counted as a character when used. The password used for interactive logins cannot be "quit," "help," "HELP," or "?" because these have special meaning to the password processor. If you enter a password of "quit," the login attempt is terminated. Typing a password of "help," "HELP," or "?" produces an explanatory message, and the request for your password is repeated. (See Section 2, in the *New Users' Introduction to Multics, Part I*, Order No. CH24).

pathname
A character string that specifies a segment by its position in the directory hierarchy. The pathname can be relative or absolute (see below). (See "Pathnames" in Section 3.)

pathname (absolute)
A concatenation of a segment's entryname with all superior directories leading back to the storage system root. (See "Pathnames" in Section 3.)

pathname (relative)
A pathname that names a segment in its relation to the working directory. (See "Pathnames" in Section 3.)

person name table (PNT)
System table containing all Person_ids (persons and fictitious persons) registered on Multics with their encoded password, default project, address, and certain other data.

Person_id
A unique name assigned to each user of the system. It is usually some form of the user's name and contains both uppercase and lowercase characters. It may not contain blank characters. Associated with the Person_id is a single password. The Person_id and the password can be used to identify a person on several projects. (See Section 2 in the *New Users' Introduction to Multics, Part I*, Order No. CH24).

Person_id.mbx
A message segment used to convey messages between processes. (See the print_mail and accept_messages commands in the Commands manual.)

physical volume
A disk pack. Sometimes the combination of pack and disk drive is referred to as the physical volume.

pointer
An address value. On Multics, an address consists basically of a segment number and an offset within the segment.

primary name
The main name associated with a segment, directory, multisegment file, or link. (See the list command in the Commands manual.)

process
A program or group of programs in execution; an address space and an execution point. Each logged-in user a has process. (See "Process" in Section 1.)

process directory
A directory containing those segments that are meaningful only during the life of a process. These segments include the stack(s), free storage, PIT, and various temporary segments.

process initialization table (PIT)
The segment (in the process directory) that contains information about process initialization, i.e., Person_id and Project_id, home directory, attributes, and accounting data. See the user_info_ description in the Subroutines manual.

process overseer
A procedure called during process initialization that sets up the environment. It then calls the listener to start reading commands.

project
An arbitrary set of users grouped together for accounting and access control purposes.

project administrator
A person who has the access to specify spending limits and other attributes for the users on a particular project.

project definition table (PDT)
An administrative data base that defines all people authorized to use an account.

project master file (PMF)
An ASCII file giving the names, attributes, and account limits of the users on a particular project. It is compiled into a project definition table.

Project_id
The name assigned to a project.

pure procedure
: A procedure that does not modify itself.

quit request
: Several commands that read input from the keyboard use the typed request "quit" or "q" to indicate to them that the user is done. This is *not* the same as issuing the quit signal.

quit signal
: A method used to interrupt a running program. The quit condition is raised by pressing the key on a terminal, such as ATTN, BRK, INTERRUPT. This condition normally causes the printing of QUIT followed by establishment of a new command level. (See "System Conditions" in Section 7.)

quote
: A character used to delimit strings in commands and source programs. On Multics this is the double quote, ASCII octal 042, not to be confused with the single quote or apostrophe, octal 047.

raw access
: Also referred to as the raw mode, it is the access mode granted a process to an object by discretionary access control. Raw access to an object is computed from the access control list (ACL), ring brackets, and AIM attributes of the object. (See discretionary access control.)

ready message
: A message that is printed each time a user is at command level. Printing this message may be inhibited, or the user may define his or her own ready message. The standard system ready message tells the time of day, the number of CPU seconds and pages of information brought into main memory since the last ready message, and the current listener level (if greater than 1).

reconnect.ec
: An exec_com segment, prepared by the user and stored in the home directory, that is invoked automatically when the user connects to a disconnected process. It is often used to execute commands such as set_tty, to ensure that terminal modes are what the user desires them to be. When a terminal is disconnected, none of its modes, set by default or by set_tty commands, are remembered. This is because the old modes may not be appropriate to the terminal that is used to reconnect to the process. Thus, if the user typically sets various terminal modes in the start_up.ec, or by explicit command, it may be helpful to make a reconnect.ec, which also sets these modes. For the same reason that the system does not retain modes across process disconnection, the user should not automatically set modes that may conflict with the characteristics of any of the terminals that might be used. An example of a device-independent terminal characteristic that the user might choose to alter is line-editing characters.

record
: (1) The smallest unit of disk allocation, containing 1024 36-bit words (4096 characters).
: (2) In PL/I and FORTRAN, a block of data transferred during input or output.

recursion
: The ability of a procedure to invoke itself.

reference name
: When a segment is made known to a process, a symbolic name may be associated with it for the duration of that process. This is called initiation. By default this is the file system entryname found by the linker when searching for a program. Reference names need not be the same as any of the segment's entrynames. (See "Reference Names" in Section 3.)

relative pathname
: See pathname (relative).

retrieval
: The process of copying a segment or directory back into the directory hierarchy from backup tapes. This is normally done by the operations staff using Retriever.SysDaemon at the request of the user. (See Section 8, "Backup.")

ring
: A particular level of privilege at which programs may execute. Lower numbered rings are of higher privilege than higher numbered ones. The supervisor program runs in ring 0, most user programs run in ring 4. (See Section 6.)

ring brackets
: A set of integers associated with each segment that define in what rings that segment may be written, read, called, or executed. (See Section 6.)

root
: The directory that is the base of the directory hierarchy. All other directories are subordinate to it. It has an absolute pathname of >. (See Section 2, "Storage System.")

scheduler
: See traffic controller below.

search rules
: A list of directories that are searched to find a command, subroutine, or data item referenced symbolically. Each directory is examined, in order, to find the given external name. Search rules are not used when a segment is addressed by its pathname, which explicitly specifies the directory containing the segment. (See "Search Rules" in Section 2.)

segment
: Basic unit of information within the Multics storage system. Each segment has access attributes, at least one name, and may contain data, programs, or be empty (null). (See "Segments" in Section 1.)

shriek names
: See unique names below.

snap (to snap a link)
The process of finding that segment (and entry point in the segment) that is referenced by a link pair and replacing the link pair with a pointer to that entry point. This is part of the dynamic linking mechanism, by which external symbolic references (subroutine calls, PL/I external variables, FORTRAN common blocks) are resolved while the program is running.

standard service system (SSS)
A group of commands and subroutines that are provided as part of the standard Multics system. They are located in the directories `>system_library_standard` and `>system_library_1`. (See Section 2, "Storage System.")

stack
A pushdown list where active procedures maintain private regions used for temporary variables and interprocedure communication. (See "Stack Header" and "Stack Frames" in Section 4.)

standard entry type
A storage system entry type that is created in ring 0 is a standard entry type. Specifically, segments, links, multisegment files, data mangement files, and directories are the standard entries in the Multics file system.

star convention
A method used by many commands to specify a group of segments and/or directories using one name (a star name). (See "Star Convention" in Section 3.)

start_up.ec
An exec_com segment, prepared by the user and stored in the home directory, that is invoked automatically when the user logs in. It is often used to execute commands such as read_mail, abbrev, and accept_messages. Start_up exec_coms can also be written for an entire project or site, to serve either as a default start_up.ec for users who do not have their own or as an additional start_up.ec that executes for all users when they establish a new process. These are placed in either the project directory or the directory `>sc1`.

status
(1) command for printing attributes of a directory entry
(2) one of the access modes on directories
(3) a coded state word returned by peripheral devices
(See status code below.)

status code
A value returned by a subroutine indicating either the success of or the reason for failure to accomplish the requested action. Associated with standard system error codes are certain predefined messages that tell what happened. (See "Status Codes" in Section 8.)

subsystem
A collection of programs that provide a special environment for some particular purpose, such as editing, calculation, or data management. It may perform its own command processing, file handling, and accounting. A subsystem is said to be closed if:

1. all necessary operations can be handled within the subsystem

2. no way exists to use the normal Multics environment from within the subsystem

suffix
The last component of an entryname with multiple components (components are separated by a period (.)) that usually specifies the type of segment, for example, pl1, mbx, and list. (See Appendix E, "List of Names with Special Meanings.")

switch
A path in the I/O system through which information is sent. (See attach and detach above and Section 5, "Input and Output Facilities.")

SysDaemon
The Project_id with which most of the system daemons login to perform their functions. See daemon above.

system administrator
A person who has the access to register users, create projects, perform accounting runs, and perform other functions necessary for the administration of the system.

System Administrator Table (SAT)
a binary table specifying the projects that use the system, the privileges granted to these projects, and their project administrators.

system_control_dir (scl, system_control_1)
The directory that contains those segments and directories used to control the operation of the system including the answer table, who table, person name table, project PDTs, etc.

terminal ID
: A character string that identifies a particular terminal at an installation.

terminal type
: A character string that identifies the terminal device, e.g., TN300, for one similar to the GE TermiNet 300. The terminal type is associated with the user's terminal and/or the modes associated with terminal input/output.

terminate
: The opposite of initiate: to delete reference names for a segment. This is sometimes done to substitute one version of a command or subroutine for another that had been known to the process. (See "Reference Names" in Section 3.)

traffic controller
: The module in the system that determines when a process is to run and how long it will run. It also notifies processes of events that have occurred such as timers, I/O events, and signals from other processes.

translation (translator)
: The process of compiling a source language program or data base into an object segment. (See "Creating an Object Segment" in Section 4.)

unique name (shriek name, exclamation point convention)
: A name, generated from a system clock value, that is guaranteed to be different from any other name so generated (e.g., !BBBnZNlqLQddRJg).

upgraded directory
: See directory (upgraded) above.

user_dir_dir (udd)
: The user directory directory, which contains all project directories. Its pathname is >udd, and all user segments and directories are subordinate to it. (See "Pathnames" in Section 3.)

User_id
: A character string representing a user or group of users (also referred to as "access identifier"). It consists of three components: Person_id.Project_id.tag. A User_id is often used as an argument to a command. Depending on the specific command, sometimes all the components are not specified (for example, the tag component is often omitted). The star convention may be used, also depending on the command being invoked. (Refer to the relevant command description in the Commands manual to see if the command in question accepts these conventions.)

volume backup
: A backup system which operates at the physical storage volume level. It provides physical volume rebuilding in the event of a failure, as well as segment and subtree retrieval.

volume label
: A label on each physical volume that identifies that volume to the system.

VTOC
: Volume table of contents. Each physical volume contains a VTOC containing information about the segments on that volume.

who table (whotab)
: A segment that contains a list of users who are currently logged in together with certain attributes such as log in time, load, and terminal type.

wired segment
: A portion of the system that (of necessity) remains resident in the main memory at all times; e.g., page control, teletype buffers, etc.

word
: A hardware-related unit of information that on Multics is 36 bits.

working directory (working_dir)
: See directory (working) above.

SECTION 2

MULTICS STORAGE SYSTEM

The basic unit of storage in the Multics storage system is the segment. Segments form a tree-structured data base that is organized by a hierarchy of directories. As shown in Figure 2-1, any segment or directory can be located by its entry in the directory immediately superior to it. That directory is located in the same manner by its entry in a superior directory and so on, up to the root of the tree. The immediately superior directory is also referred to as either the containing or parent directory.

SEGMENT REFERENCES

All segment references begin at the root of the tree and consist of a string of entrynames ending with the name of the target segment. Such a string of entrynames is called an absolute pathname. The greater than character (>) is used to separate entrynames and is also used at the beginning of the pathname (by convention, the root directory is never explicitly specified). In Figure 2-1 the absolute pathname for the segment named "chess" is:

```
>udd>Others>Jones>chess
```

The syntax of entrynames and pathnames is given in detail in "Entrynames" and "Pathnames" in Section 3.

LOGICAL VOLUMES

Segments in the storage system hierarchy are stored on disk volumes. These disk volumes are organized into groups called logical volumes. A logical volume consists of one or more disk volumes used by the storage system to contain segments. Storage is allocated on logical volumes according to the following rules so that, generally, related segments will reside on the same logical volume.

1. All segments immediately inferior to a particular directory reside on the same logical volume.

2. When a directory is created, the logical volume on which its inferior segments will reside is set; and this attribute cannot be changed except by deleting and re-creating the directory. The logical volume is the same as for the new directory's parent unless a master directory for a logical volume is being created by a special call (master directories are described later in this section).

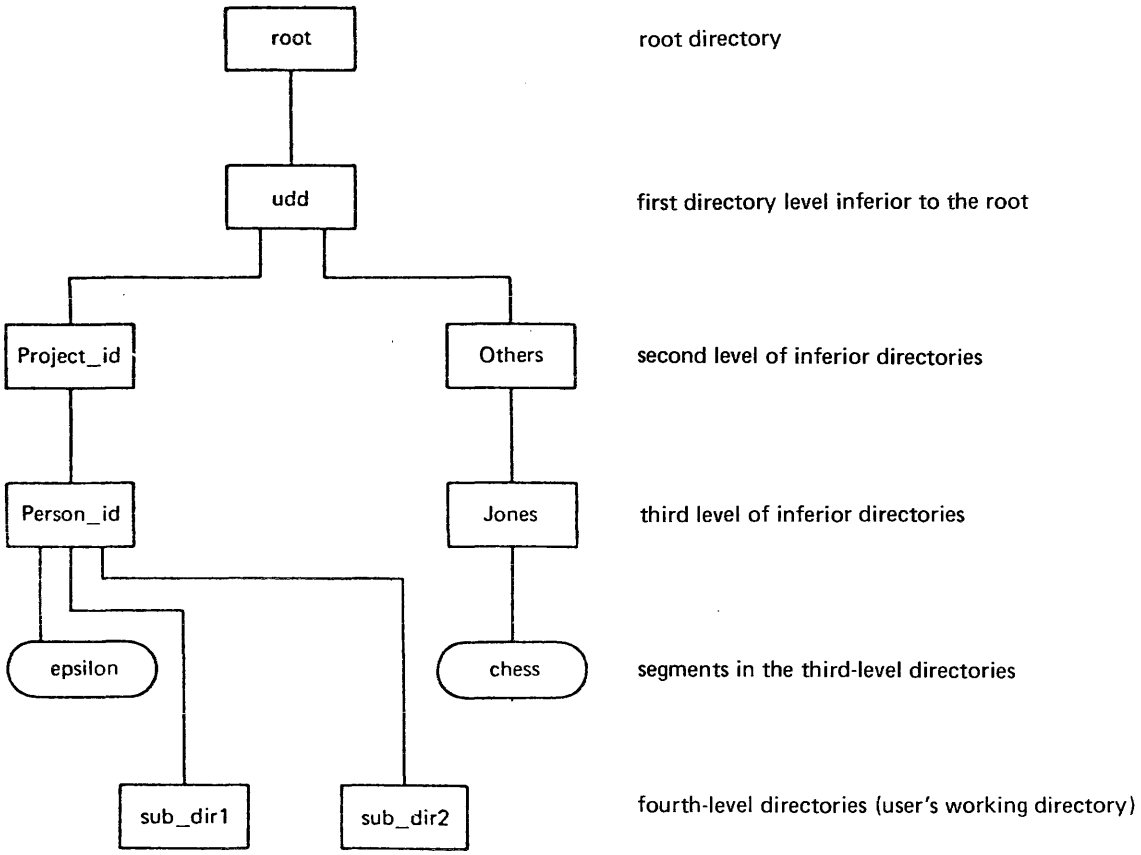


Figure 2-1. Storage System Hierarchy

When a logical volume is created, a registration record is created for it. This record contains the following information:

- list of the physical disk volumes comprising the logical volume
- owner identification
- public or private switch
- list of master directories
- list of users with quota accounts

Each Multics system has a special logical volume, called the root logical volume, that contains all directories in the storage hierarchy. The root logical volume is always mounted (this means that all the disk volumes comprising it are mounted) and the information contained on it is always available. The segments themselves may not be mounted all the time, since all but the root logical volume can be be mounted and demounted by the operator at the user's request. Segments are available to users only when the logical volume on which they reside is mounted.

Logical Volume Attachment

Segments stored on public logical volumes can simply be referred to when needed by a process. When a segment is stored on a private logical volume, a process must attach the volume (via the attach_lv command) before it can use the segment. The volume remains attached until the process explicitly detaches it (via the detach_lv command). Many processes can use such a segment simultaneously, but each process individually must attach the logical volume. Access to the volume is checked when the attach request is made. The user must have rw access to the access control segment (ACS) associated with the volume. The ACS is maintained by the volume's owner. If a process attempts to use a segment on a private logical volume that it has not attached, an error indication will be returned by the system.

Master Directories

When a new directory is created, its segments will, by default, reside on the same logical volume as the segments of its parent directory. If the segments in the new directory are to reside on some other logical volume, a master directory must be created (this is done using the -logical_volume control argument of the create_dir command). A master directory is simply the point where the hierarchy "branches out" to another logical volume. Figure 2-2 illustrates the relationship among directories, master directories, and logical volumes.

All master directories for a given logical volume are listed by name in the registration record for the volume. A master directory possesses attributes in the same manner as other directories except that quota for the master directory is not drawn from the quota account of the containing directory but from a master directory quota account maintained in the logical volume registration record. In order to create a master directory on a logical volume, the user must have a quota account on the logical volume with unallocated quota sufficient to satisfy the request.

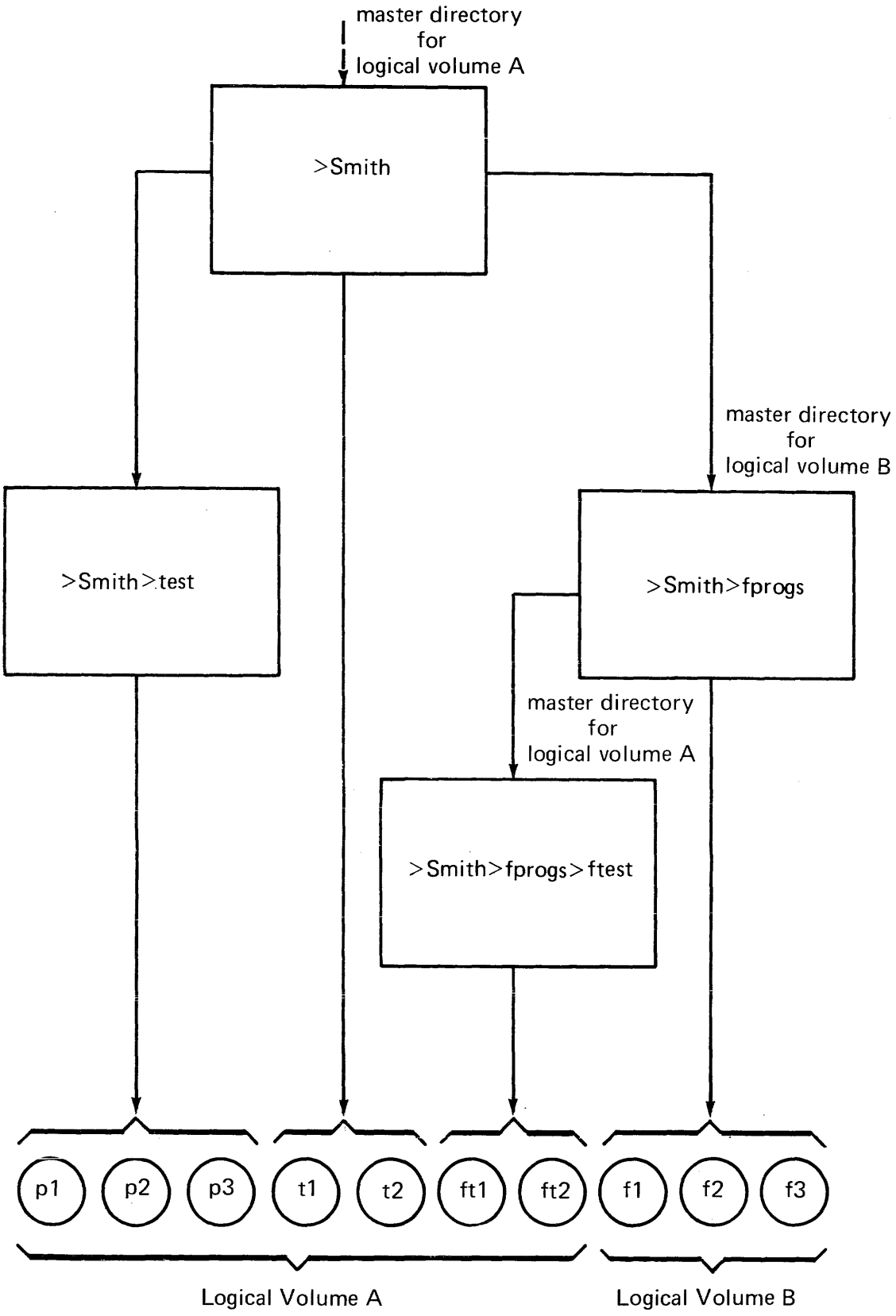


Figure 2-2. Relationship of Directories to Logical Volumes

STORAGE SYSTEM ENTRY TYPES

The basic elements within the Multics storage system are segments and directories. Multics supports additional entry types that are maintained for convenience or to aid programmers who require a storage medium with special qualities or attibutes. The various entry types are described below.

Segment

The segment is the unit of storage of the Multics System that is analogous to a file on other systems. A segment is a collection of instructions or data specified by a user.

Directory

A directory is a catalog of subordinate entries.

Link

A link entry is a reference to an entry in another directory. The reference is made by giving the pathname of the target entry.

Multisegment File

Very large data bases may exceed the size of a single segment. In such cases, Multics treats this data base as a group of segments in a single multisegment file. The segments are grouped under a common directory whose multisegment file indicator is set. The directory and its contents are called a multisegment file (MSF).

Any directory whose multisegment file indicator is not 0 is an MSF. For an MSF, this indicator is a count of the number of segments it contains. Not all of the attributes listed above are applicable to MSFs. Some of the attributes are the same for any entry: however, due to the nature of an MSF when viewed as a file, many of the attributes are implemented differently. For example, the bit count of an MSF is the sum of the bit counts of the segments it contains. The access control list for an MSF directory applies to all of the segments it contains. The safety swtich attribute can be used; however, if it is set for one of the segments in the MSF, it should be set for all of them. For more specific information on these and other attributes of MSFs, refer to the msf_manager_ subroutine in the *Multics Subroutines and I/O Modules* manual, Order No. AG93.

Most standard system programs that work on segments also work on MSFs. However, some commands and subroutines will give unpredictable results when used on MSFs. The programmers should consult the individual command or subroutine description before invoking it on an MSF.

Data Management File

A data management (DM) file is composed of a set of pages known as control itervals, numbered from 0 through N and adddressable only through software calls to the file manager. Data is accessed by specifying a control interval number, byte offset, and length.

Data management files can be implemented with concurrency control and recovery support. At present the ability to use data management files is available only to programs accessing files through the Multics Relational Data Store (MRDS) facility.

Extended Entry Types

The Multics storage system supports special-case entry types, called extended entry types. The following system-supplied storage system elements have been implemented as extended entries: mailboxes, forum meetings, message segments, before journals, and the person name table. These entry types are called extended entry types because the Multics storage system has been enhanced (extended) to treat these storage elements as segments (even though they are structured differently than segments). Each of the extended entry types is identified by a suffix appended to the entry name, as described below:

Name	suffix
mailbox	.mbx
forum meeting	.forum
message segment	.ms
before journal	.bj
person name table	.pnt

Users should note that the extended entry types specified above are those available with the Honeywell-supplied system. User-created file system objects can also be implemented as extended types, if desired. See Section 4 for information on the creation of extended entries.

ENTRY ATTRIBUTES

Entry attributes are listed below. Not all of the attributes listed below are applicable to all entry types.

A process may perform explicit modification of an attribute by calling a standard storage system subroutine. Implicit modification is automatic and occurs as a result of some change to the target entry. For example, when data is written into an existing segment, the date-time contents modified attribute is changed.

access class *

The access class of an entry is established when the entry is created. It is used to restrict access to users who meet specific security requirements. The access class attribute cannot be modified. Access class characteristics are described in detail in Section 6. *

* access control list

The access control list (ACL) maintains a list of access names, specifying classes of users who are allowed access to the entry and, for each class, the mode of access permitted. The access specified may be null, indicating that no access is permitted. The ACL attribute is used in conjunction with the access class attribute to determine access rights when a particular process refers to the * entry. An ACL can be explicitly modified. See Section 6 for a complete discussion of access control.

* author

The author attribute of an entry is the access identifier of the process that created the entry. This attribute cannot be modified.

* bit count

| The bit count attribute gives the length (in bits) of the entry. The bit count can be modified by any process with write access to the segment and is maintained by the user rather than the system. Any procedure that modifies the segment length should also modify the bit count since many system commands and subroutines depend on its accuracy.

* bit count author

The bit count author attribute contains the access identifier of the process that last set the bit count. This attribute is automatically updated when the bit count is set.

* complete volume dump switch

| This attribute controls whether an entry is to be dumped during a complete volume dump of the physical volume on which it resides. The default is to dump. This attribute should only be disabled if the data can be easily recreated. This attribute is unavailable for segments on private logical volumes unless the process has attached the logical volume.

* copy switch

| The copy switch is used to determine whether the entry itself or a copy of the entry is made available when an attempt is made to modify the segment. If the copy switch is "off," all processes share the original entry and only those processes with write access can perform write operations. If the copy switch is "on," the above holds true for the original entry, but processes without write access are automatically given their own copies of the entry. These copies (made in the process directory) can be modified and are retained for the life of the process.

current length *

The current length attribute gives the length in pages of an entry. This attribute is modified by the system when data is stored beyond the existing current length or when the entry is truncated. This attribute is not available for entries on private logical volumes unless the process has attached the logical volume.

damaged switch *

This attribute controls a switch that curtails access to entries damaged by a device error or system crash. When damage occurs, the entry should be inspected to determine whether its contents can be recreated or the entry retrieved. The damaged switch is automatically set to "off" when an entry is truncated to zero words. This attribute is unavailable for entries on private logical volumes unless the process has attached the logical volume.

date and time dumped *

This attribute records the time at which a backup copy of the entry was last made by the hierarchy dumper. This attribute is unavailable for entries on private logical volumes unless the process has attached the logical volume.

date and time contents modified *

This attribute records the approximate time at which the contents of the entry were last modified. The date-time-contents-modified (DTCM) attribute of a non-directory entry (except for f, below) is set whenever the transparent-modify switch (in the ASTE) is off, and:

(a) the entry is created,
(b) the entry is truncated,
(c) the DTCM is the subject of an inquiry and the entry has been modified since the last time the DTCM was set,
(d) the entry is deactivitated after having been modified since the last time the DTCM was set,
(e) the last in-core page of the entry drifts out of memory and the entry was modified since the last time the DTCM was set,
(f) the segment is a directory and an inferior (active) entry has its DTCM set for any of the above reasons, or
(g) the hierarchy or volume reloader calls to explicitly set DTCM.

Except for privileged operations, the DTCM is set to the current time. The DTCM is set in the VTOCE for non-active entries and in the ASTE for active entries. Whenever the VTOCE is updated from the ASTE, the ASTE DTCM value is placed in the VTOCE.

The dumpers and reloaders set the transparency switches for entries in order to insure that the DTCM values reflect the values that were dumped. The transparent-modify switch is always set for directories, so that only directory DTCM is updated when:

(a) an entry DTCM is updated for any entry below the directory, or
(b) an operation occurs that updates the date-time-entry-modified attribute of any entry below the directory.

* date and time entry modified

This attribute records the last time any attribute of the entry was modified. It is implicitly updated after any modification.

* date and time used

This attribute records the last time the target entry was referenced. The date-time-used (DTU) attribute is set whenever the transparent-use switch (in the ASTE) is off, and:

(a) the entry is created,
(b) the entry is the subject of an inquiry and has pages in memory,
(c) the entry is deactivated with pages in memory,
(d) the hierarchy or volume reloader calls to explicitly set the DTU, or
(e) the last in-core page of the entry drifts out of memory.

Except for privileged operations, the DTU is always set to the current time. The DTU is set only in the VTOCE for non-active entries and in the ASTE for active entries. Whenever the VTOCE is updated from the ASTE, the ASTE DTU value is placed in the VTOCE.

The dumpers and reloaders set the transparency switches for entries in order to insure that the DTU values reflect the values that were dumped.

date and time volume dumped

This attribute records the time a volume dumper process dumps the entry contents to tape.

* dnzp switch

The "don't null zero pages" (dnzp) switch is used to control how pages of a an entry that contain only words of zeroes are represented on disk. If the switch is off (i.e., null zero pages), then pages that contain all zeroes are not actually written on the disk and are not charged against quota; instead, they have a "null" address placed in their file map. If the switch is on (i.e., don't null zero pages), then a page of zeroes is treated just like any other page and is written to disk and charged against quota.

entry point bound *

The entry point bound attribute provides a way of limiting which locations of a gate entry may be targets of a call. The hardware does not permit an inward call to the entry if the word number specified in the call is equal to or greater than the entry point bound word number.

incremental volume dump switch *

This attribute controls whether an entry is to be dumped by the volume dumper during an incremental dump cycle. The default is to dump. The incremental volume dump should be distinguished from the complete volume dump; this switch can be turned off with relative safety for things that are seldom modified. This attribute is unavailable for entries on private logical volumes unless the process has attached the logical volume.

initial access control lists *

An ACL is created for each new entry in a directory by copying the initial ACL from the containing directory. The initial ACL contains default values (see Section 6 for these) and can be explicitly modified by any process that has modify access to the directory at validation level. No access to the containing directory is required. *

logical volume identifier *

The logical volume identifier for a directory identifies the logical volume to be used for entries created in a particular directory. Its value is either inherited from the parent directory or explicitly set by supplying the -logical_volume control argument to the create_dir command. The value of this attribute cannot be changed after the directory is created.

logical volume identifier *

The logical volume identifier for a segment names the logical volume on which the segment's contents are stored. Its value is set when the entry is created and cannot be modified.

master directory switch *

The master directory switch indicates whether or not a directory is a master, one whose entries reside on a different logical volume than those of its parent directory. The switch is turned on when the directory is a master directory and turned off when it is not. *

* maximum length

The maximum length attribute sets a limit on the size an entry can attain. Maximum length is accurate to units of 1024 words. The maximum value in words is 255K (K = 1024). This attribute is not available for entries on private logical volumes unless the process has attached the logical volume.

* multisegment file indicator

This attribute is used to indicate that the directory is associated with a multisegment file. The value of the attribute is the number of segments (components) in the file. The multisegment file indicator is implicitly modified by multisegment file primitives when the number of components of the file changes. The user can also modify it by using the set_bit_count command.

* names

Each entry can have many names. The first name returned by the storage system is called the primary name. For more information on names, see "Entrynames" in Section 3.

* quota

The quota attribute gives the maximum number of storage records permitted to entries in a particular directory.

* records used

The records used attribute gives the amount of secondary storage (in records) occupied by the entry. This attribute is implicitly modified when there is any change to the number of nonzero records used. This attribute is not available for entry on private logical volumes unless the process has attached the logical volume.

* ring brackets

The ring brackets attribute is used in connection with other access control mechanisms to determine access rights to the target entry. See Section 6 for a complete discussion of ring brackets.

safety switch *

The safety switch attribute is used to protect an entry from deletion. If the safety switch is set, the user is asked if the entry should be deleted before a deletion command or request is executed on the entry.

security out-of-service switch *

When this switch is on, the directory in which it occurs and all inferior entries cannot be referenced. The switch is automatically set when an access class discrepancy is detected. This attribute can only be modified by a system security administrator.

type *

The type attribute indicates the entry type. The type attribute cannot be modified.

unique identifier *

The unique identifier attribute is a number assigned when an entry is created to distinguish it from all other entries in the storage system. This attribute cannot be modified.

use count *

This attribute is a count of the number of page faults taken on an entry since its creation. This attribute is unavailable for segments on private logical volumes unless the process has attached the logical volume.

*

SYSTEM DIRECTORIES

A single directory hierarchy is used for both system and user segments. Figure 2-3 shows, at the upper level of the storage hierarchy, the basic structure assumed by the Multics system. Additional segments and directories can be created at this level of the structure as well as at lower levels.

As shown in Figure 2-3, several system directories are contained in the root. These are always present and are described below.

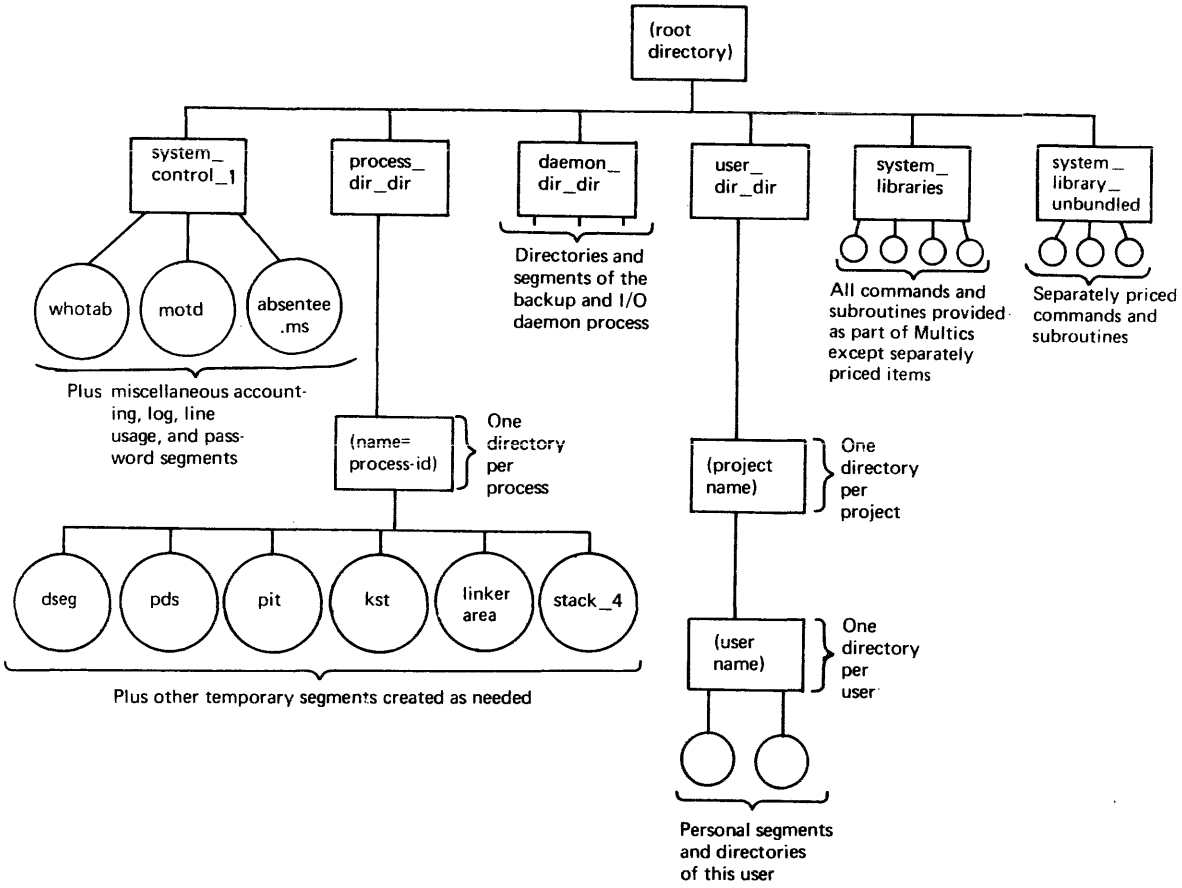


Figure 2-3. Directory Hierarchy

1. system_control_1
   This directory contains information associated with system accounting, user authorization, and logging-in procedures. Project administration tables are stored in a directory subtree beginning at this directory. The following three segments are the only generally accessible ones entered in system_control_1: the table printed by the who command; the message of the day; and absentee queue segments.

2. process_dir_dir
   This directory contains a process directory for each currently active process. The name of an individual process directory is derived from the unique identification of the process. A process directory contains temporary segments created by a process and retained only for the life of that process.

When a process is created, a process directory is established with the six initial segments described below:

process data segment (PDS)
: A supervisor data base, the PDS keeps a record accessible only to the supervisor.

known segment table (KST)
: A supervisor data base, the KST contains the correspondence between segment numbers and segments known to the associated process. This segment is accessible only to the supervisor.

process initialization table (PIT)
: The PIT contains information that is used to initialize the process.

descriptor segment (DSEG)
: A supervisor data base, the DSEG contains the correspondence between segment numbers and their absolute memory addresses and access permissions. This segment is accessible only to the supervisor. It is always segment 0.

stack_n
: This segment contains the stack used for PL/I automatic variables and for subroutine call and return operations. One stack segment is created for each active ring; the last character of the stack name is the ring number. The ring 0 stack, unlike the others, is not kept in the process directory; it is kept in system_library_1. Actually, there are a number of these stacks in ring 0, and they are multiplexed among running processes.

unique_name.area.linker
: This segment, managed by linker, contains interprocedure links and PL/I internal static storage. If the total requirements for linkage information and static storage exceed the length of a segment, additional segments are created as needed under a similar name. In addition, each active ring has its own linkage segment. In addition, the linkage area also contains other system storage (e.g., external static and fortran common).

Other segments that are created by various Multics subsystems and editors are also commonly found in the process directory.

3. daemon_dir_dir

This directory contains segments that support system daemon processes, such as automatic file backup and bulk (card and printer) input and output. The queues of the I/O facilities are the only generally accessible segments in this subtree.

4. user_dir_dir
This directory is the beginning of a tree containing all segments belonging to individual users. It contains entries for a set of directories, one for each project. Each project directory generally contains one personal directory (home directory) for each user associated with that project. Individual users can create their own directories, inferior to their own personal directory.

5. system_libraries
The standard Multics commands and subroutines are combined in the following system libraries:

```
system_library_standard
system_library_1
system_library_tools
system_library_unbundled
system_library_obsolete
```

The procedures in these directories are documented in the Commands and Subroutines manuals. A library of unbundled software (system_library_unbundled) may also be present. Unless the user specifies otherwise, these directories (except for system_library_obsolete) are included in the list of directories to be searched during dynamic linking. See "Dynamic Linking" and "Search Rules" in Section 4.

SECTION 3

NAMING, COMMAND LANGUAGE, AND TERMINAL USAGE

CONSTRUCTING AND INTERPRETING NAMES

The various types of names used on Multics are constructed and interpreted according to certain definite, fixed conventions. The names discussed below are entrynames, pathnames, star names, equal names, reference names, offset names, command names, subroutine names, condition names, request identifiers (IDs), and I/O switch names. User names are discussed under "Access Control" in Section 6 since they are primarily used to specify access control information.

Entrynames

An entryname is the name of an entry (segment, directory, etc.) in the file system. An entryname consists of at least one nonblank and no more than 32 ASCII characters. Any entry can have more than one entryname. In general, entrynames consist of uppercase and lowercase alphabetic characters, digits, underscores (_), and periods (.). The underscore is used to simulate a space for readability; e.g., a segment might be named delta_new. (Including a space in an entryname is permitted, but is cumbersome, since the command language uses spaces to delimit command names and arguments.) The period is used to separate components of an entryname, where a component is a logical part of the name. Null components (i.e., zero length components) should not exist. A null component results if its name begins or ends with a period or contains two adjacent periods. Several system conventions (e.g., the star convention and equal convention both described below) operate on components. Also, compilers implemented on Multics expect the language name to be the last component of the name of a source segment to be compiled, e.g., square_root.pl1 for the name of a PL/I source segment. See "Program Preparation" in Section 4 for details on programming conventions.

Only the greater than (>) character is prohibited in entrynames, since it is used to form pathnames as described below. Since standard commands attach special meanings to them, several other characters are not recommended for entrynames, including the less than (<), asterisk (*), question mark (?), percent sign (%), equal sign (=), dollar sign ($), quotation mark ("), two consecutive colons (::), vertical bar (|), and parentheses characters. In addition, all ASCII control characters (e.g., space, tab, carriage return, etc.) are not recommended for use in entrynames because some of these characters have a special meaning in the command language, and the others are hard to use (they do not print out correctly and are difficult to type). Non-ASCII characters are not permitted in entrynames.

Pathnames

A pathname is a sequence of entrynames. Each entryname except the last in a pathname is the name of a directory entry (or link to a directory entry) in the storage system hierarchy. (See "Directory Contents" in Section 2.) The last entryname in a pathname is the name of an entry (segment, directory, etc.). The last entryname in a pathname can be a starname or an equalname, or the pathname may be followed by an archive component name, an offset, or a symbolic definition. Each entry in the hierarchy has an entry in a superior directory.

Any entry can be found by following the appropriate entries from a designated directory through inferior directories. The length of a pathname must not exceed 168 characters. An absolute pathname traces an entry from the root directory; a relative pathname traces an entry from the current working directory.

An absolute pathname is formed from a sequence of entrynames, each preceded by a greater than character. Each greater than character denotes another level in the storage hierarchy. The entryname following the initial greater than character designates an entry in the root directory (see Figure 3-1, below). An example of an absolute pathname is:

```
>udd>Project_id>Person_id>epsilon
```

The directory named user_dir_dir (udd) is immediately inferior to the root; Project_id is an entry in udd; Person_id is an entry in Project_id; and epsilon is an entry in Person_id. Each intermediate entry in the chain can be either a directory or a link to a directory. The final entry, epsilon, can be a directory, a segment, a multisegment file, a link, a data management file, or one of the extended entry types. A maximum of 16 levels of directories is allowed from the root to the final entryname.

A relative pathname looks like an absolute pathname except that it does not contain a leading greater than character, and can begin with less than characters as explained below. It is interpreted by commands as a pathname relative to the user's working directory. The simplest form of relative pathname is the single name of an entry in the user's working directory. For example, in Figure 3-1, the relative pathname beta refers to the entry beta in the user's working directory sub_dir2. On a slightly more complex level, the relative pathname my_dir>omega refers to the entry omega in the directory my_dir, which is immediately inferior to the user's working directory sub_dir2.

A less than character can be used at the beginning of a relative pathname to indicate that the directory immediately superior to the working directory is where the following entryname is to be found. The less than character can be used to denote levels in the storage hierarchy similar to the use of the greater than character. Each less than character represents one level up the hierarchy (toward the root), starting at the current working directory. In this way, a directory several levels superior to the current working directory can be searched for the first entryname in the relative pathname.

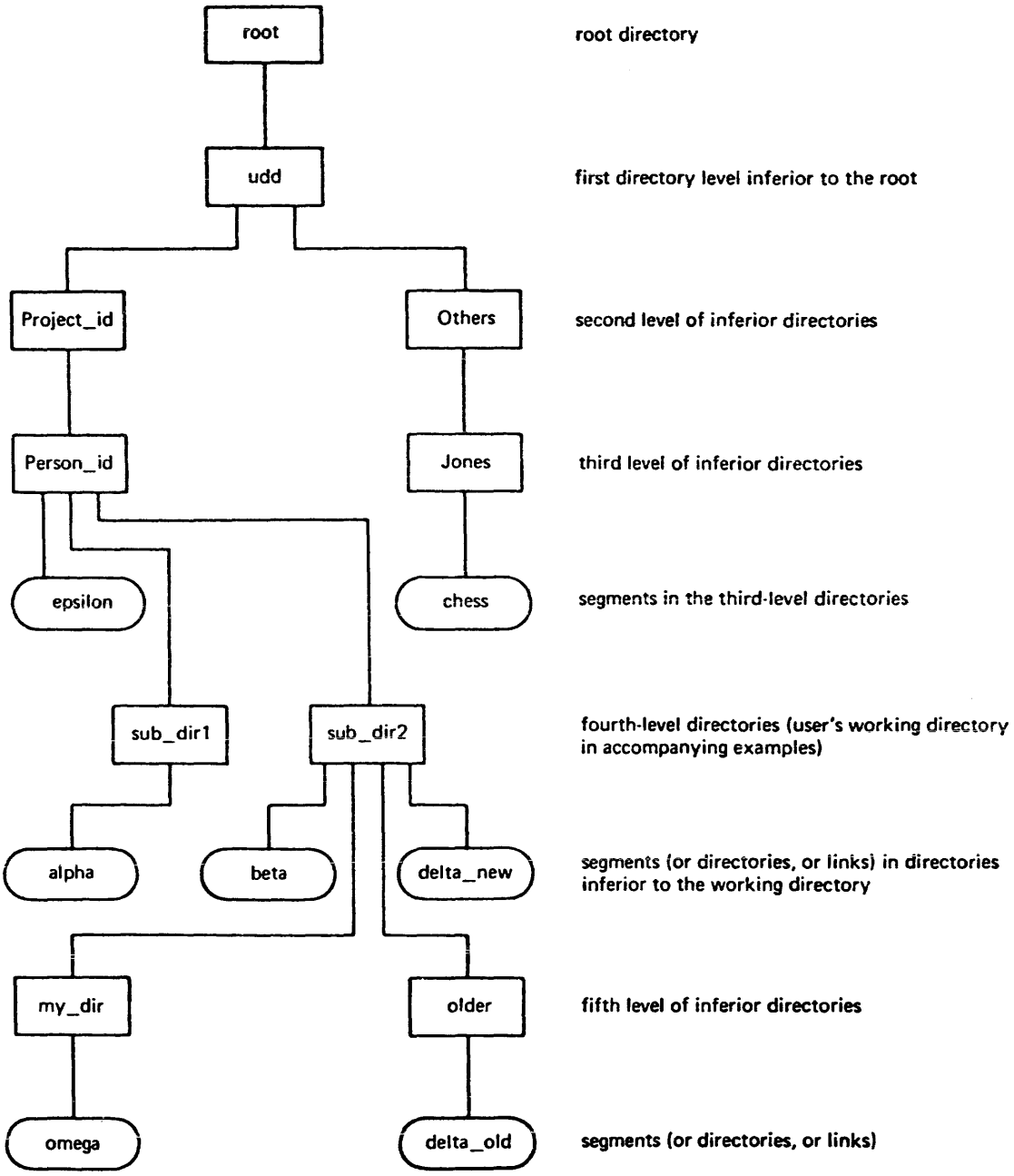


Figure 3-1. Sample Storage Hierarchy

The following examples (using the sample hierarchy in Figure 3-1) show some relative pathnames and the absolute pathnames of the segments they identify when the user's working directory is:

```
>udd>Project_id>Person_id>sub_dir2
```

Relative Pathname	*Segment*
delta_new	>udd>Project_id>Person_id>sub_dir2>delta_new
older>delta_old	>udd>Project_id>Person_id>sub_dir2>older>delta_old
<sub_dir1>alpha	>udd>Project_id>Person_id>sub_dir1>alpha
<<<Others>Jones>chess	>udd>Others>Jones>chess

Archive Component Pathnames

If the final component of a pathname contains a "::" sequence, it is not interpreted as the name of an entry, but rather as a specification of an archive name and the name of a component in the archive. The string preceding the "::" sequence is interpreted as the archive name; the suffix ".archive" need not be specified, since it is assumed. The string following the "::" sequence is interpreted as the name of the component in the archive. Only one "::" sequence may appear in the final component of a pathname. The component name may be up to 32 characters in length; the name of the archive may be up to 32 characters including the ".archive" suffix, or 24 omitting it.

Example:

```
print >udd>Sample>Smith>source::blank.pl1
```

This command prints the component blank.pl1 in archive >udd>Sample>Smith>source.archive.

NOTE:

Not all commands are prepared to manipulate archive components, and hence do not interpret this syntax. If an archive component pathname is given to a command that cannot manipulate archive components, the error message:

Archive component pathname not permitted.

is printed. In particular, commands that create or write into segments generally do not implement this syntax. To determine whether a particular command implements this syntax, consult the Commands manual description.

Star Names

Many commands accept starnames to identify the entities to be examined or operated upon. Starnames are names containing wildcard characters used to specify sets of entities or to facilitate typing. The star convention defines the wildcard characters and matching criteria.

Starnames are constrained by the application. Commands that use starnames to match file or directory names permit the final entryname in a pathname to be a starname. In this case, the starname is also an entryname and is subject to the restrictions on entrynames such as the 32 character limit. Such a command would match the starname against the names of all of the entries in the directory (as determined from the pathname) and select those entries of appropriate type which have at least one matching name.

A starname matches links if the command utilizing the star convention operates on the link itself. In general, commands do not work upon the targets of links matching a starname. Similarly, a starname matches every entryname of an entry if it operates on name attributes. Otherwise, an entry is generally selected only once even if it has several names matched by the starname.

RULES FOR CONSTRUCTING STAR NAMES

1. A starname is a character string.

2. A starname is made up of components. Components are delimited by the beginning and end of its name, and by the period (.) character, referred to as a dot.

3. Each question mark (?) character appearing in a starname is treated as a special character.

4. Each asterisk or star (*) character appearing in a starname is treated as a special character.

5. Each occurrence of two consecutive asterisks (**), called a doublestar, appearing in a starname is treated specially.

6. Each component consisting only of a doublestar, called a doublestar component, is treated specially.

7. Three or more consecutive asterisks (***) are invalid.

INTERPRETING STAR NAMES

A starname is compared to a set of names; names that satisfy the following criteria are considered to match the starname.

1. If the starname contains no special characters (stars or question marks), then the rules for PL/I string comparison are used.

2. Trailing ASCII space characters are not significant.

3. Each nonspecial character matches itself literally. The matching constructs must be in one-to-one correspondence between the starname and the matched name, in the same order.

4. Each question mark matches exactly one character within a component, so it matches any single character except dot.

5. Each star matches any number of characters within a component, so it matches any number (including zero) of any character except dot.

6. Each doublestar matches any number of characters, including zero.

7. Each doublestar component matches any number of entire components, including zero. Note that the dot or dots delimiting the doublestar component match component boundaries, and if zero components are matched, they match the same boundary. The boundary can be a dot or the beginning or end of the matched name.

The following examples illustrate some common forms for star names.

!????????????

identifies all 15 character one-component entries beginning with ! (called unique names because such names are generated by the unique_chars_ subroutine, described in the Subroutines manual, and by the unique active function) in the user's working directory.

ad?

identifies all three-character one-component entries in the user's working directory that begin with ad.

ad?*

identifies all one-component entries in the user's working directory that begin with ad and have three or more characters.

*

identifies all one-component entries in the user's working directory.

*_data

identifies all one-component entries whose first component ends with _data preceded by any number of other characters (including none).

.

identifies all two-component entries in the user's working directory.

*.pl1

identifies all two-component entries in the user's working directory that have pl1 as their second component.

prog*.pl1

identifies all two-component entries whose first component begins with the letters prog followed by any number of other characters (including none), and whose second component is pl1.

`sub_dir>my_prog.new.*`
identifies all three-component entries in the directory sub_dir (which is immediately inferior to the user's working directory) that have my_prog.new as their first and second components.

interest_*_data.*.*
identifies all three-component entries whose first component begins with interest_, ends with _data, and has any number of characters (including none) in between.

*.**.my_seg
identifies all entries with two or more components of which the last is my_seg.

**
identifies all entries in the user's working directory.

**.pl1
identifies all entries with pl1 as the last (and possibly only) component.

my_prog.**
identifies all entries with my_prog as the first (and possibly only) component.

`sub_dir>prog?.**.pl1`
identifies all entries in the directory sub_dir (which is immediately inferior to the user's working directory) with two or more components, of which the first component has exactly five characters and begins with prog, and the last component is pl1.

foo
identifies any one-component entries in the user's working directory that has the substring "foo" in their name.

**foo*
identifies any name in the user's working directory which has the substring "foo" in its last component.

*.**.**.**
identifies all entries in the user's working directory. Any name which contains at most one single star component, at least one doublestar component, and nothing else will match anything.

Equal Names

Some commands that accept more than one pathname as their arguments allow the entrynames of pathnames following the first, or source, pathname to be equalnames. This is generally to be expected if the command allows the source entryname to be a starname. An equal name is an entryname containing special characters that represent one or more characters from the entryname (or entrynames, when a star name is used) that corresponds to it. Commands that accept equal names provide a powerful mechanism for mapping certain character strings from the first pathname into the second pathname of a pair. Use of the equal convention reduces the typing required for the second pathname, and it can be essential for mapping character strings from entrynames identified by a star name into the equal name, because these character strings are not known when the command is issued.

CONSTRUCTING EQUAL NAMES

An equal name is constructed according to the following rules:

1. An equal name is an entryname. Therefore, it is composed of a string of 32 or fewer ASCII printing graphics or spaces, none of which can be the greater than (>) character. Unlike an entryname, an equal name cannot contain control characters such as backspace, tab, or newline.

2. An equal name is composed of one or more nonnull components. This means that an equal name cannot begin or end with a period (.) and cannot contain two or more consecutive periods.

3. Each percent sign (%) character appearing in an equal name component is treated as a special character.

4. Each equal sign (=) appearing in an equal name component is treated as a special character.

5. An equal name component consisting only of a double (==) or triple equal sign (===) is treated as a special component.

6. An equal name containing four or more consecutive equal signs is illegal.

INTERPRETING EQUAL NAMES

An equal name maps characters from the entrynames that match the star name (first entryname) into the second entryname of a pair according to the following rules:

1. Each percent sign (%) in an equal name component represents the single character in the corresponding component and character position of the corresponding entryname. An error occurs if the corresponding character does not exist.

2. An equal sign (=) in an equal name component represents the corresponding component of a corresponding entryname. An error occurs if the corresponding component does not exist. An error also occurs if an equal sign appears in a component that also contains a percent character. Only one equal sign can appear in each equal name component, except for a double or triple equal sign component, as noted in the next two rules.

3. The double equal sign (==) component of an equal name represents all components of an entryname that have no other corresponding components in the equal name. Often, the double equal sign component represents more than one component of the corresponding entryname. If so, the number of components represented by the entire equal name is the same as the number of components in the entryname. When the equal name contains the same number of components or more components than the entryname, a double equal sign is meaningless and, therefore, ignored. (See the examples below.) Only one double equal sign component can appear in an equal name.

4. The triple equal sign (===) component of an equal name represents the entire corresponding entryname. The triple equal sign component is used to add components to a name (see below). Only one triple equal sign component may appear in an equal name and no other component of that equal name may contain percent signs or equal signs.

The rules above impose no restrictions on the form of the entrynames identified by the equal name. These names can contain null components. However, the rename and add_name commands cannot be called with arguments that contain null components, because these commands treat their arguments as either star names or equal names. The -name control argument of the rename and delete_name commands can be used to change or delete entrynames that contain null components, control characters, or other characters reserved by the star, equal, archive, or virtual pointer conventions. (See the Commands manual for descriptions of the rename and delete_name commands.) Entrynames present in directories have usually been subjected to validity tests for entrynames and thus are usually valid.

This page intentionally left blank.

The following examples illustrate how equal names might be used in rename and add_name commands.

First, the single equal sign. The command:

```
rename  random.data_base  ordered.=
```

is equivalent to:

```
rename  random.data_base  ordered.data_base
```

and the command:

```
add_name  world.data  =.statistics  =.census
```

is equivalent to:

```
add_name world.data  world.statistics world.census
```

The command:

```
rename  random.data.base =.=
```

is equivalent to:

```
rename  random.data.base  random.data
```

The star convention is used in the command:

```
rename  *.data_base  =.data
```

to rename all two-component entrynames with data_base as their second component so these entrynames have, instead, a second component of data.

The command:

```
rename  program.pll  old_=.=
```

is equivalent to:

```
rename  program.pll  old_program.pll
```

and the command:

```
add_name  data  first_=_set
```

is equivalent to:

```
add_name  data  first_data_set
```

An error would be produced by the command:

```
rename  alpha  beta.=.gamma
```

because the first entryname of the pair does not contain a component corresponding to the equal sign in the second name.

Next, the double equal sign. In the two examples that follow, the first entryname has components that correspond to the double equal sign in the equal name of each pair. As a result, the number of components represented by the equal name is the same as the number of components in the first entryname. The command:

```
rename  one.two.three  1.==
```

is equivalent to:

```
rename  one.two.three  1.two.three
```

and the command:

```
add_name  one.two.three.four.five  1.==.5
```

is equivalent to:

```
add_name  one.two.three.four.five  1.two.three.four.5
```

In the example that follows, the equal name contains the same number of components as the entryname. Therefore, the double equal sign does not correspond to any components of the entryname and is ignored. The commands:

```
rename  alpha.beta  ==.x.y
rename  alpha.beta  x.y.==
rename  alpha.beta  x.==.y
```

are all equivalent to:

```
rename  alpha.beta  x.y
```

In the next example, since the equal name contains more components than the entryname, the double equal sign corresponds to no components of the entryname and is ignored. The command:

```
add_name  able  ==.baker.charlie
```

is equivalent to:

```
add_name  able  baker.charlie
```

The command:

```
add_name  **.ec  ==.absin
```

uses the star convention to add a name to each entry with an entryname whose last (or only) component is ec. The last component of this new name is absin instead of ec, and the first components (if any) are the same as those of the original name ending in ec (e.g., the name alpha.absin would be added to the entry named alpha.ec).

The command:

```
rename foo.test.pl1 ==.old
```

is equivalent to:

```
rename foo.test.pl1 foo.test.old
```

With the triple equal sign, this command becomes:

```
rename foo.test.pl1 ===.old
```

and is equivalent to:

```
rename foo.test.pl1 foo.test.pl1.old
```

because the triple equal sign represents the entire corresponding entryname. For the same reason, the command:

```
add_name alpha.** ===.1
```

adds the name "alpha.1" to "alpha", "alpha.pl1.1" to "alpha.pl1", etc.

Note that a triple sign component in an entryname implies that the new name will have more components than the old name. This is different from a double equal sign component as can be seen if the command:

```
add_name alpha.** ==.1
```

is used instead. The latter command has a different effect. For example, it attempts to add the name "alpha.1" to both "alpha.pl1" and "alpha.list", leading to a name duplication error.

The command:

```
rename  ???*.data  %%%.=
```

renames all two-component entrynames that have a last component of data and a first component containing three or more characters so that the first component is truncated to the first three characters and the second component is data (e.g., alpha.data would be renamed alp.data). The command:

```
rename  *.data  %%%.=
```

results in an error if the first component of any name matching *.data has fewer than three characters.

Archive Component Pathnames and Equal Names

Some commands that accept pairs of pathnames as their arguments (e.g., the compare_ascii command described in Commands manual) allow either or both of the pathnames to be archive component pathnames (e.g., source::blank.pl1 as described in an earlier part of this section). The first pathname may contain star names as part of the archive name (that portion of the pathname before the double colon (::)) and the component name (that portion appearing after the double colon). The second pathname may also be an archive component pathname, and it may use equal names in the archive name and the component name. This usage makes it easy for the user to request an operation on one or more components in an archive and either segments or components in other archives where the segment/component names are constructed from the original component name using equal names. Additionally, if the operation is to be performed on two components, the name of the second archive may also be derived from the name of the first archive using equal names. The two portions of an archive component pathname, the archive name and the component name, are treated separately by the star and equal name conventions. Thus, for instance, a triple equal sign used in the archive name will append only the corresponding archive name, not the entire pathname. The command:

```
compare_ascii source.ll::my_data ===.ud::my_data
```

is equivalent to:

```
compare_ascii source.ll::my_data source.ll.ud::my_data
```

The rules for constructing and interpreting equal names in the two portions of an archive component pathname are identical to those for ordinary equal names described earlier in this section.

The following rules are used to determine whether to apply a given equal name to the archive or component name specified in the first (source) pathname when constructing the second (target) pathname:

1. If neither the source nor target pathnames are archive component pathnames, an equal name in the target pathname is applied to the source pathname just as described in the previous section on equal names.

2. If the source pathname is not an archive component pathname, but the target pathname is an archive component pathname, an equal name in the target component name is applied to the source entryname; an equal name is not permitted in the target archive name in this case.

3. If the source pathname is an archive component pathname, but the target pathname is not an archive component pathname, an equal name in the target entryname is applied to the source component name, not the source archive name.

4. If both the source and target pathnames are archive component pathnames, an equal name in the target archive name is applied to the source archive name, and an equal name in the target component name is applied to the source component name. When applying the equal name to the source archive name, ".archive" suffix is removed from the archive name.

The following examples illustrate the use of archive component pathnames and equal names in the compare_ascii command.

The command:

```
compare_ascii test.pll source::old.===
```

is equivalent to:

```
compare_ascii test.pll source::old.test.pll
```

and compares the segment "test.pll" with the component "old.test.pll" in the archive "source.archive".

The command:

```
compare_ascii source.s::print_data.pll ===.ud::===
```

is equivalent to:

```
compare_ascii source.s::print_data.pll source.s.ud::print_data.pll
```

and compares the component named "print_data.pll" in the two archives, "source.s.archive" and "source.s.ud.arhive".

The command:

```
compare_ascii my_prog.pll ===::his_prog.==
```

is invalid because there is no archive name in the source pathname corresponding to the "===" in the target pathname.

Finally, the command:

```
compare_ascii tools::my_prog.pll his_prog.==
```

is equivalent to:

```
compare_ascii tools::my_prog.pll his_prog.pll
```

Reference Names

A reference name is a name used to identify a segment that has been made known by the user. Initiating a reference name for a segment is one way to make a segment known to the user's process. (See "Making a Segment Known" in Section 4 and "Process" in Section 1.) A segment can be made known via the initiate command (described in the Commands manual) and the hcs_$initiate and hcs_$initiate_count subroutines (documented in the Subroutines manual). When a segment is made known and a reference name initiated for the segment, its reference name is entered into the reference name table. If the user uses the initiate command to initiate a reference name for a segment, the reference name need not have any similarity to the entryname of the segment. For example:

```
initiate >udd>Project_id>Person_id>debug newdebug
```

makes the segment named debug in the user's home directory known with the reference name newdebug.

A segment can be addressed by its reference name either from command level or from within a program. When a segment is addressed, the hcs_$make_ptr subroutine (described in the Subroutines manual) uses search rules to locate the desired segment. By default, the first search rule is "initiated_segments", causing the reference name table, listing reference names for segments, to be searched first. If the segment has not been made known and a reference name has not been initiated for the segment, the search continues until a segment with an entryname that matches the reference name is found. (Search rules are described in detail under "Search Rules" in Section 4.)

A reference name is associated only with segments made known in a process. The same reference name can be used in two different processes to refer to two different segments. Also, a reference name/segment binding exists only for the duration of the process in which it is specified. It is possible to break that binding by making the segment unknown, thus causing all external references (links) from other segments to the unknown segment to be unsnapped and causing the segment to no longer be known in the process (by any reference name). Any reference name of an unknown segment can be used again in the process to refer to a different segment. (See the descriptions of the terminate and terminate_refname commands in the Commands manual and the term_, hcs_$terminate_file, and hcs_$terminate_seg subroutines in the Subroutines manual.) For example, there is a system command named debug. If the user has made a segment in his home directory known with the reference name debug, every time he calls debug he gets the version in his home directory rather than the system provided version of debug. If the user wants to call the system version of the command, he must first make the segment in his home directory unknown.

A user must keep his search rules in mind when he initiates and terminates reference names. For example, if a user has initiated the reference name debug for a segment in his home directory and he also has a segment named debug in his working directory, every time he calls debug he gets the version in his home directory. If he wishes to use the version of debug in his working directory, he must first terminate the reference name debug for the segment in his home directory. Future calls to debug then find the version in the user's working directory unless home directory

appears before working directory in his search rules. If this is the case, the user must explicitly initiate the reference name debug for the segment in his working directory.

Individual reference name/segment name bindings can be terminated in a process without making the segment unknown unless the reference name removed is the only one on the segment. (See the descriptions of the terminate_single_refname command in the Commands manual and the term_, hcs_$terminate_name, and hcs_$terminate_noname subroutines in the Subroutines manual.) If a user has called the system version of the debug command and later wants to make known the version of debug in his home directory with the reference name debug, he must first terminate the reference name to the system version. For example:

```
terminate_single_refname debug

initiate >udd>Project_id>Person_id>debug debug
```

causes calls to debug to invoke the routine in >udd>Project_id>Person_id with one exception: other system routines bound together with debug (via the bind command described in the Commands manual) continue to invoke the system routine since those links were presnapped when the routines were bound together. The terminate, terminate_single_refname, and terminate_refname commands and the term_ subroutine unsnap dynamic links, whereas the hcs_ entry points (described in the Subroutines manual) do not unsnap links.

Entry Point Names

Procedures frequently have more than one entry point, and data segments frequently have internal locations that are known externally by symbolic names. The names of entry points and internal locations are generically called entry point names. Each designates symbolically an offset within a segment. The location specified can be referred to by the construction ref_name$entry_point_name where the dollar sign separates the reference name and entry point name.

In many cases the entry point to a procedure has the same name as the segment itself (or the segment has several entrynames corresponding to the names of its entry points). A shorthand notation allows the entry point name to be assumed to be the same as the reference name. For example:

```
call square_root (n);
```

x is interpreted to mean:

```
call square_root$square_root (n);
```

and the command line:

```
rename a b
```

is equivalent to:

```
rename$rename a b
```

If the user has renamed a procedure segment (perhaps to preserve an old copy) or created a storage system link to a segment using a different name, the full reference name/entry point name construction must thereafter be used when referring to that segment as a procedure or external data segment. For example, a PL/I subroutine compiled with subr_name as the label of its procedure statement and then renamed new_name must be referred to as new_name$subr_name.

Command, Subroutine, Condition, and I/O Switch Names

These types of names all have some conventions in common.

- Each is required to be 32 characters or less in length.
- All ASCII characters are legal in any position except as noted in the following points and "Entrynames" above.
- System subroutine names end in an underscore to prevent conflicts with subroutine names given by users. Users can easily avoid conflicts by not having an underscore as the last character of any of their subroutine names.
- Condition and I/O switch names that are part of the system, according to the new convention, end in an underscore to help prevent conflicts with names given by users. See the appendix entitled "List of Names with Special Meaning" for a list of previously established condition and I/O switch names that do not end in an underscore.
- Command and subroutine names should not contain a period; i.e., they should have only one component.

Request IDs

Several system facilities operate by having users enter requests in one or more queues. These requests are processed by the system at some later time. Examples of such facilities include the absentee facility and the I/O daemon facility. (See the descriptions of the enter_abs_request, dprint, and dpunch commands in the Commands manual.)

There are a number of commands that operate on requests that are already in the queues, for example to list them, cancel them, or change their priority. These commands need to identify a particular request. Often the pathname or entryname of the segment associated with the request is sufficient to uniquely identify it. However, sometimes several requests are associated with the same segment, and there can be other reasons why the segment name alone is an unsuitable identifier of the request.

In these cases the request ID may be used to select one request from a group. The request ID is based on the date and time at which the request was entered into the queue. The full request ID is a 19 character decimal number, of the form YYMMDDhhmmss.ffffff, giving the year, month, day, hour, minute, second, and 6-digit fractional second (in Greenwich Mean Time) at which the request was entered. In many cases, the six digits to the immediate left of the decimal point (hhmmss) are unique among all requests currently in the system. Often, fewer than six digits are unique. When requests are entered in rapid succession by a single command line, they can have the same seconds digit in their IDs, and the tenths digit is required for uniqueness. (If a user tries to get a request ID using the clock command or the date_time_$format subroutine, he must specify -zone GMT to get a valid request ID.)

The 8 character ID, hhmmss.f, is printed by default by the request listing commands; printing of the longer ID can be requested by a control argument. The commands that take a request ID argument accept any number of digits, before or after the decimal point (with a decimal point being assumed after the rightmost digit if none is typed). The ID is accepted provided that it matches only one request in the group being selected from. If it matches more than one, the user is told how many, and instructed to supply more digits of the ID.

Date/Time Names

Multics use of date/time values is described in the following subsections. Multics accepts dates from the year 0001 through 9999. The Julian calendar is used for dates from 0001-01-01 through 1582-10-04. The Gregorian calendar is used for dates from 1582-01-15 through 9999-12-31. (The dates from October 5, 1582 through October 14, 1582 do not exist; they were dropped when the Gregorian calendar was adopted.) The leap day is always February 29. The lower limit on dates of January 1, 0001 A.D., is used since it begins the era; the upper limit of December 31, 9999, was chosen to limit year numbers to four digits. The time zones as now defined are used regardless of the year. The Multics date/time software does not account for "leap seconds", and, therefore, the difference between any two binary clock values that are precisely an integral number of days (hours, minutes, seconds, etc.) apart is guaranteed to be evenly divisible by the number of microseconds in a day (hour, minute, second, etc.).

DATE/TIME INPUT VALUES

Often the user must supply date and time information to a command. Programs that accept date and time information use the convert_date_to_binary_ subroutine (see the Subroutines manual) to convert a time string to an internal (binary) value.

Time Strings (DT Values)

The time string can have up to six parts: adverbial offset, date, time, day of week, signed offset, and time zone. Adverbial offsets, if present, must appear leftmost in the string. Beyond that, all the parts are optional and can be in any order. The parts can be made up of alphabetic fields, numeric fields, and special characters.

An alphabetic field is made up of letters and must contain a whole word or an abbreviation (often made up of the first three letters of the word). No distinction is made between uppercase and lowercase characters. Although this description gives examples in English, each of the words is available in several languages. Any of these languages can be used in time strings, but all words within a given string must be in the same language. To see the languages defined on a site, the user can type

```
display_time_info -lang
```

A numeric field consists of an optionally signed integer of one or more decimal digits. The special characters that can be used in either alphabetic or numeric fields are: the slash (/), the period (.), the colon (:), the plus (+), the minus (-), and the comma (,). Blanks are not required between alphabetic and numeric fields in the time strings; however they are required between two numeric fields unless the second field begins with a plus (+) or minus sign. For example,

```
2days4hours10minutes
1245.17+7hours
10/17/79Wednesday
```

Unless otherwise indicated in the command description, the input time string must be specified as a single argument. This means quotations must enclose time strings that contain spaces. Alternatively underscores are used instead of blanks in the time string. For example,

```
09/25/79__1442.6_+5_hours
```

Usually when entering a time string, the time zone is omitted. Although the time zone is seldom seen, it is very important: it determines the interpretation of items given in the time string; it is also involved in defaults supplied for missing items. All defaults are taken from the current absolute time, adjusted by a working time zone. If a zone is specified in the string, that becomes the working zone; otherwise the process default time zone is used.

This means that whether the user converts a string with an explicit zone, such as "XXXX_ast", or sets the process default to "ast" and then converts the string "XXXX", the same absolute time is returned. (Note that setting the process default also influences output conversion, while giving an explicit time zone does not.) To display the default zone, the user can type

```
print_time_defaults zone
```

The six parts of the time string are described below. In these descriptions whenever an assumed value is mentioned, it refers to the current date/time adjusted to the working zone.

1. date
   is the day of the year; it can be specified only once. The user can supply a date using normal date format, calendar date format, day of the week, date keywords, fiscal week, request-id, or can omit it entirely. If no date is present, it is assumed to be the next occurrence of the time specified; for instance, "10A" gives the date on which 10:00am next occurs. If no date and time is given, the current date is used.

In normal date format, the user can specify dates as month (or month abbreviation), day of month, and year; or as day of month, month, and year. The year is optional and, if omitted, is assumed to be the year in which the date occurs next; that is, if today is March 16, 1985, then March 20 is equivalent to March 20, 1985; while March 12 is the same as March 12, 1986. There are three forms of normal date:

```
16 March   16 March 1985
March 16   March 16 1985   March 16, 1985   (The comma is optional)
3/16       3/16/85         3/16/1985
```

The calendar date format allows the user to supply dates as a year, month, and day of month, separated by minus signs. This is the International Standards Organization (ISO) standard format. The year is required, and the user can give it as a year of the century. For example,

```
85-12-31  or  1985-12-31
```

represents December 31, 1985.

The day of the week is a date specifier if present with no other form of date. It then selects the first occurrence of the named day after today. The date keywords are "yesterday", "today", and "tomorrow"; for instance,

```
6:35A today
yesterday +120days
```

The fiscal week is of the form FWyyyyww. FW is the fiscal indicator (in English), yyyy is the year number, and ww is the week number. The fiscal week begins on Monday and ends on Sunday. This form converts to the date of Monday, but the user can select a day within the week by adding a day name; for example, "FW198413 m" gives "03/26/84 0000. Mon", while "FW198413 m Wed" gives "03/28/84 0000. Wed". The user can separate the fiscal indicator from the number, but the ordering must remain, i.e., "FW185425" or "FW 185425", but not "185425 FW".

A request-id is a 19-character string used by several programs in the system, such as list_output_request. It contains a complete date from year, in century, down through microseconds in this form

```
yymmddHHMMSS.SSSSSS
```

If the user provides no zone, it is interpreted in GMT, not the process default. A request-id specifies a time as well as a date, so the user can give no other time specification.

2. day of week
is a day of the week (e.g., Monday) and can be present only once. When the day of the week is present along with one of the other forms of date specification, that date must fall on the indicated day of the week.

3. time

is the time of day and can only be present once. If omitted, it is assumed to be the current time. The user can give time as 24-hour format, 12-hour format, or the time keyword "now". The 24-hour time format consists of a four-digit number followed by a period: hhmm., where hh represents hours and mm minutes. The user can follow this number by an optional decimal fraction-of-a-minute field (e.g., hhmm.m). Also acceptable are hours and minutes fields separated by colons (hh:mm). The user can optionally follow this by either a fraction-of-a-minute field (hh:mm.m) or a seconds field (hh:mm:ss). The seconds, in turn, can include a fraction-of-second field (e.g., hh:mm:ss.s). Examples of 24-hour time are:

```
154.1.5
1545.715
15:45
15:45.715
15:45:42
15:45:42.08
```

The user must end the 12-hour time format with a meridiem designator (i.e., A, P, am, pm, noon (n), midnight (m)). The user can indicate midnight and noon by giving just the meridiem designator. The user can precede the designator by time expressed as hours, hours:minutes, or hours:minutes:seconds (including an optional fraction of a second or fraction of a minute). Examples of 12-hour time are:

```
midnight
5 am
5:45A
3:59:59.000001pm
11:07:30.5pm
12 n
```

There is a set of illegal times--24:00-24:59--which are handled anyway. These are taken to mean 00:00-00:59 of the following day: midnight (00:00) is the beginning of a day, not the end.

4. signed offset

is an adjustment to be made to the clock value specified by the other fields. The user can supply offsets in any the following units:

year	years	yr
month	months	mo
week	weeks	wk
day	days	da
hour	hours	hr
minute	minutes	min
second	seconds	sec
microsecond	microseconds	usec

Each unit can be present one or more times, each preceded by an optionally signed fixed point number. If offset fields are the only thing present, the offsets are added to the default values of date and time, as described above.

If the month offset results in a nonexistent date (e.g., "Jan 31 3 months" would yield April 31), the last date of the resulting month is used (i.e., April 30). Examples of offset fields are:

```
3 weeks -60 hours  (60 hours before 3 weeks after now)
1.5 hr 5min  (an hour and 35 minutes from now)
1 hour 5 minutes  (an hour and five minutes from now)
```

The order in which offset values are applied to the clock value can affect the resultant clock value. Offset values are applied in the following order:

```
year, month, week, day, hour, minute, second, microsecond
```

"Monday 6 am 2 weeks" means "two weeks after the next occurrence of Monday, at 6:00 am on that day".

Assuming that today is September 25, 1985, then

```
10/1 -1 day +1 month
```

results in a clock value for 10/31/85, rather than for 10/30/85.

Note: There is also a nonoffset use of these words, available in combination with the word "this". These combinations are not forms, but can be used in building date and time parts. For example, "this_month_1,_this_year" or "this_hour:23" is valid, while just "this_day" is not. The exact form of this combination varies according to the language used. In some languages the word for "this" changes according to the gender of the unit it is applied to; in others there may be a single word that does the job.

5. adverbial offset
is a before/after kind of adjustment that can be used any number of times. The user can recognize it by the presence of "before", "on", or "after" in the time string. If present, it must appear first. These are the forms available:

```
DAY-NAME before
DAY-NAME on or before
DAY-NAME before or on
DAY-NAME after
DAY-NAME on or after
DAY-NAME after or on
SIGNED-OFFSETs before
SIGNED-OFFSETs after
```

When adverbial offsets are present, they partition a time string into a series of adjustments followed by a base time. These sections are processed from right to left. The example below has 3 sections: first "6:00 am 400sec" is handled, supplying all necessary defaults and making the ordinary (400sec) offset adjustment; then "Monday after" is applied to give a new value; finally "2 wk -5min after" is applied to this new value to give the final value.

```
2 wk -5min after Monday after 6:00 am 400sec
20 minutes before now
2 days after today
2500 weeks after 1776-7-4
Tue after Mon on or after 11/1
```

The last item describes election day in the USA: the first Tuesday after the first Monday in November.

6. zone
is the time zone to be used in making the conversion to Greenwich mean time, which is the internal form of all clock readings. It can be either a zone differential or any of the zone abbreviations known at your site. A zone differential is a five-character string, "sHHMM" (s is a sign, HH is a two-digit hour, and MM is a two-digit minute).. The user can use this only immediately following a time specification: "12:15-0330" says that 12:15 is the local time, and -0330 specifies that the local time was generated by subtracting 3.5 hours from GMT. To list the zone abbreviations known at a site, type:

```
display_time_info -zones
```

If any defaults are needed, the current instant is broken down into years, months, days, and so forth with respect to a "working zone". This working zone can make much difference because, for example, at a given instant it can be Tuesday in New York and Wednesday in Bankok, or it can be 22:07 in London and 3:37 in Singapore. Thus the zone is as important for week days and years as it is for hours and minutes.

Many of the date/time commands allow you to supply a "-zone X" argument. In this case, X can be any of the zones known at a site; it can't be a time differential.

DATE/TIME OUTPUT VALUES

One important way to get a clock value into a readable form is by the date/time commands (calendar_clock, day, etc). The first argument to the clock command is a control string describing the format wanted. All other date/time commands have intrinsic formats. Command use of clock values is really going from a readable form to a readable form without keeping the internal value.

By convert_date_to_binary_, an input time string is converted to internal form. This is the usual form for storing dates in data bases. To convert one of these into a readable form, the user can call upon date_time_ to get a 24-character form like this:

```
03/14/79  0000.0 cet Fri
```

But when other formats are needed, date_time_$format is available. It takes a clock value and a control string describing the format wanted and returns a string ready for printing.

An effort has been made to make all date/time outputs from the system software be usable as date/time inputs to system software, but the control string mechanism is so flexible that the user can easily use it to generate formats that are not recognizable. Also there are strings that are apparently recognized, but which are ambiguous, for instance, "7/1/82"; in the United States this means the 7th month, first day, but in many European countries this would mean the 7th day of the first month. Multics follows the American interpretation.

Time Format

The control string to date_time_$format is either a keyword or a character string consisting of text and/or selectors. The selectors are always identified by a leading circumflex character (^). There are two types of selectors: ^<keyword>, which allows a keyword to be embedded within a format, and the general form ^XX. XX is a two-letter code that specifies what information is wanted. The user can place an optional PL/I picture specification between the ^ and XX if the default form is not adequate. If the control string does not contain any circumflex characters, it must then be one of the known set of keywords. Each keyword identifies a control string for a predetermined format named by that keyword.

List of Format Keywords

all

^9999yc-^my-^dm__^Hd:^MH:^99.(6)9UM^zd_^za_^da ^fi ^(6)9fw ^ma dy^dy dc^dc Uc^Uc.

calendar_clock

^9999yc-^my-^dm__^Hd:^MH:^99.(6)9UM_^za_^da.

clock

^9999yc-^my-^dm ^Hd:^MH:^99.(6)9UM ^za ^da.

date
is the process default value for date.

date_time
is the process default value for date and time.

iso_date
^9999yc-^my-^dm.

iso_date_time
^9999yc-^my-^dm ^Hd:^MH:^SM ^za.

iso_long_date
^9999yc-^my-^dm ^da.

iso_long_date_time
^9999yc-^my-^dm ^Hd:^MH:^99.(6)9UM ^za.

iso_long_time
^Hd:^MH:^99.(6)9UM.

iso_time
^Hd:^MH:^SM.

multics_date
^my/^dm/^yc.

multics_date_time
^my/^dm/^yc ^Hd^99v.9MH ^xxxxza^xxxda.

multics_time
^Hd:^MH.

request_id
^yc^my^dm^Hd^MH^99.(6)9UM.

Note: The output from this keyword is specified in the process default time zone. To obtain a valid request ID, specify -zone GMT when you supply the request_id keyword. See "Request IDs" in this section.

system_date_time
is the system default value for date and time.

system_date
is the system default value for date.

system_time
is the system default value for time.

time
is the process default value for time.

A site can change the "system" keywords. For an application that depends upon the historic formats used in Multics, the three builtin "multics" keywords are available.

Processing of a control string proceeds by scanning the control string until a circumflex is found or the end of the string is reached. Any text (including any blanks) passed over is copied to the output string. The selector is then interpreted and executed. This causes a datum from the input clock value to be edited into the output string. Processing continues in this way until the control string is exhausted.

The user can express dates and times placed in the output string in units of years, months, weeks, days, hours, minutes, seconds, and microseconds, and the total calendar value as a single unit; for example the user could express the calendar value representing 79-09-08 9:42A GMT as 1979 years, as 722702 days, or as 722702.112499 days. This is the set of "total" selectors:

```
^yc total number of years in the calendar value ^mc total number
of months in the calendar value ^dc total number of days in the
calendar value ^Hc total number of hours in the calendar value ^Mc
total number of minutes in the calendar value ^Sc total number of
seconds in the calendar value ^Uc total number of microseconds in
the calendar value.
```

The user can also express dates and times as the number of units remaining after a larger unit has been removed from the calendar value; for example, 09/08/79 09:42 includes units for the 9th month of the year, the 8th day of the month, the 9th hour of the day, and the 42nd minute of the hour. The following are the most common:

```
^my   month in the year
^dm   day of the month
^dw   day of the week
^Hd   hour of the day (24-hour format)
^Hh   hour in half day (12-hour format)
^MH   minute of the hour
^SM   second of the minute
^US   microsecond of the second.
```

There are several items of date/time data that are nonnumeric, such as day of week, day of month, and time zone used for conversion.

```
^mn   month name
^ma   month name, abbreviated (char (3))
^dn   day name
^da   day name, abbreviated (char (3))
^zn   time zone name
^za   time zone name, abbreviated (char (4))
^zd   zone differential (char(5))
^mi   meridiem indicator (A or P)
^fi   fiscal indicator (FW in English)
```

The selectors of numeric data are, in general, made up of two letters taken from this sequence: c y m w d H M S U. This letters stand for calendar, year, month, week, day, hour, minute, second, and microsecond, respectively. All 81 combinations are not, however, valid. The form can generally be read as "unit of unit", e.g., "seconds of week". The first unit is always smaller than the second one. In trying to keep the specifiers reasonably mnemonic (in English) there is a problem: both month and minute begin with an "m". So all date values are used as lower-case letters while all time values are in upper case.

It is difficult to try to handle all the forms needed in a general manner. Hd is hour of the day and is thus 24-hour time; this is not always what is wanted. Hh is chosen as hour in half day to get the 12-hour form of time. To go along with this there is "mi" for Meridiem Indicator, which gives A or P to make up AM or PM. This does not give AM or PM because ANSI and ISO standards specify that time be given as "3P", not "3PM". If the user wants the M, put the literal in, e.g., "^miM".

Another way of looking at a calendar value is in terms of fiscal week. This is selected with the "^fw" code. Its value is four digits of year followed by two digits of week number, i.e., yyyyww. The default picture has been chosen to give a value of yww. The associated fiscal indicator is selected by "^fi." A complete value is obtained by specifying "^fi^fw."

The table below shows the complete set of selectors. The row specifies what unit is wanted, the column specifies within what other unit, e.g., ^Sy is seconds of year.

DATE/TIME SELECTORS

	of calendar	of year	of month	of week	of day	of hour	of minute	of second
microsecond	^Uc	^Uy	^Um	^Uw	^Ud	^UH	^UM	^US
second	^Sc	^Sy	^Sm	^Sw	^Sd	^SH	^SM	
minute	^Mc	^My	^Mm	^Mw	^Md	^MH		
hour	^Hc	^Hy	^Hm	^Hw	^Hd			
day	^dc	^dy	^dm	^dw		month	day	zone
month		^my			name	^mn	^dn	^zn
year	^yc				abbrev	^ma	^da	^za
	^Hh	<-hour of half day (12-hour form)			differential			^zd
	^mi	<-meridiem indicator						
	^fw	<-fiscal week (form: yyyyww)						
	^fi	<-fiscal indicator ("FW" in english)						

The user can control the formatting of date and time values by an optional PL/I picture specification included in the selector; for instance, a code of ^OO99yc formats the total years in the calendar value into a two-digit year of the 20th century and ^9999yc provides a full, four-digit year. The following is a brief description of the most frequently used picture characters. For more details on PL/I pictures, see the *Multics PL/I Language Specification* manual (AG94) and the *Multics PL/I Reference Manual* (AM83).

9 represents a mandatory decimal digit in the displayed value.

z represents a decimal digit in the displayed value. Nonsignificant zeros on the left are replaced by a space when they occupy a "z" digit position.

. produces a period in the displayed value. This has no relation to the location of the decimal point in the value actually being displayed. If zero suppression is in effect, this is replaced with a space.

, produces a comma in the displayed value. It has all the characteristics of the period.

v locates the value's decimal point in the result. This determines how the value digits are oriented with respect to the picture specification. If the user supplies no "v", it is assumed to appear after the rightmost picture character.

The picture characters above are sufficient for displaying most numeric values. For example, the control string ^99Hd^99.v9MH represents the time in hours, minutes, and tenth of minutes; the control string ^zz9.999vUS represents the number of milliseconds of the second, using the decimal point and "v" to scale the microsecond unit. Scaling can also be performed by a picture scale factor.

f(N) scales the value by multiplying or dividing by a power of 10, thus shifting the location of the decimal point in the value. For example, f(2) shifts the decimal two places left, effectively dividing the value by 100; f(-3) shifts three places right, effectively multiplying by 1000.

Using a picture scale factor, the user can display the milliseconds in excess of a second to the nearest tenth using the control string ^zz9.9f(3)US. You can display a value of 48634 microseconds as " 48.6" milliseconds.

There are two extensions to numeric picture handling:

Z represents a decimal digit in the displayed value. Nonsignificant zeros to the left of the decimal point are omitted when they occupy a "Z" digit position; to the right of the decimal point they are omitted when they occupy a "Z" digit position.

Z characters must appear as the leftmost or rightmost digit positions in the picture specification since these are the positions that nonsignificant zeros can occupy. Z performs a selective ltrim or rtrim (of zero) operation on the displayed value. For example, the user can specify the millisecond specification given above as ^ZZ9.9ZZUS without using a picture scale factor; with this specification you can display 48630 microseconds as 48.63 milliseconds (without the leading space or trailing zero).

O represents a decimal digit in the displayed value that is not wanted. Specifying ^99yc for a year like 1941 results in a size condition since it takes four digits to handle that number. To get the year in century the user can use ^OO99yc; this gives four digits into which the value is placed and then the first two digits are discarded. A picture like OOz9 with a value of 1502 gives 02 because the zero suppression applies to 1502, and then the first two digits are dropped.

The user can format character date/time values such as day of the week, month name, and time zone using a character picture specification with the "x" picture character.

x represents a position that can contain any character. Since national characters occur in some of the time names, avoid the "a" character. Values are left-justified in the picture specification, with truncation of the rightmost characters if the value is longer than the picture or padding with spaces on the right if the value is shorter than the picture.

For example, ^xxxxxxxxdn displays Wednesday as "Wednesday" and Monday as "Monday". The user can use a picture repetition factor to shorten the control string to "^(9)xdw". With ^(5)xmn January is displayed as "Janua" and May is displayed as "May ". (Note that in some languages the abbreviation of a time name is not the first three letters of it.)

The selector picture specification allows an extension of the "x" picture specification.

X represents an optional character position in the displayed value. The character position is omitted if there is no corresponding character in the value being displayed.

X characters must appear as the rightmost character position in the picture specification since this is the position that nonsignificant spaces can occupy. X performs a selective rtrim operation on the displayed value.

The code ^(9)Xdw displays Wednesday and Monday both without trailing spaces.

The table below shows the default picture specifications for all selectors. The row specifies what unit is wanted, the column specifies within what other unit.

DEFAULT PICTURE VALUES

	of calendar	of year	of month	of week	of day	of hour	of minute	of second
microsecond	(18)Z9	(14)Z9	(13)Z9	(12)Z9	(11)Z9	(10)Z9	(8)Z9	(5)Z9
second	(12)Z9	(12)Z9	(8)Z9	(6)Z9	(5)Z9	(4)Z9	99	
minute	(10)Z9	(6)Z9	(5)Z9	(5)Z9	(4)Z9	99		
hour	(8)Z9	(4)Z9	(3)Z9	(3)Z9	99			
day	(7)Z9	999	99	9				
month		99						
year	0099							

		month	day	zone
name		(32)X	(32)X	(64)X
abbrev		(8)X	(8)X	(8)X

99	<-hour of half day (12-hour form)
x	<-meridiem indicator
000999	<-fiscal week (form: yyyyww)
xx	<-fiscal indicator

differential s9999

The following table shows how date and times strings are displayed by a variety of control strings.

^mn ^Z9dm, ^9999yc
displays September 8, 1979.

^mn ^z9dm, ^9999yc
displays September 8. 1979.

^dm ^ma ^9999yc ^zn
displays 08 Sep 1979 Mountain Standard Time.

^my/^dm/^yc ^Hd^99v.9MH ^za ^da
displays 09/08/79 0242.4 mst Sat.

Hd:^MH:^SM^z
displays 02:42:25-0700.

9999yc-^my-^dm__^Hd:^MH:^99.(6)9UM_^za_^d
displays "979-09-08__02:42:25.048634_mst_Sat.

-^<multics_time>xyz^<multics_date>-
displays <-02:42xyz09/08/79->.

COMMAND LANGUAGE

The standard Multics command language and its conventions are described below. Various subsystems, with their own conventions, are also available (see Section 1 of this manual). In addition, user-created systems and conventions can provide specially tailored environments that supersede the following Multics conventions.

A Multics command is a system program, usually written in PL/I, that uses argument processing facilities provided by the command processor. A command name is the entryname of the segment containing that system program. Command invocation consists of a command name alone or a command name followed by character string arguments that are separated by white space (any combination of blank spaces and tabs). A command line is delimited by the newline, vertical tab, or formfeed characters, and can contain one or more command invocations separated by semicolons (;). The syntax rules for command invocations allow for such features as iteration, nesting, and function evaluation.

The Multics command processor is a mechanism for invoking programs by command name. It finds the commands by command name in the storage system hierarchy, and invokes the commands with their arguments. Commands are found via the search rules, described in Section 4. The command processor is called by the listener subroutine to process the command invocations typed by the user. The command processor can also be called from a program by using the cu_$cp entry point (see the description of cu_$cp in the Subroutines manual).

After a user logs in to the system, the listener subroutine prints a ready message on the terminal. The user is then at command level and the system is waiting for input in the form of a command line. When a user completes a command line, the command processor evaluates it. The command line obtained when all the command elements have been evaluated is called the expanded command line. It is then executed by the command processor. The listener subroutine is again invoked, and prints another ready message.

Command invocations are referred to simply as commands throughout most of the Multics documentation.

Command Environment

The command invocation consists of two basic elements: the command name and the arguments. The command name is a reference name. If followed by a dollar sign ($) and a character string, this part of the command name is an entry point name. The command processor uses the search rules of the user's process (see "Search Rules" in Section 4) to find the command in the storage system hierarchy. A pathname can be used in place of the reference name to override the user's search rules. In this case, the segment identified by the pathname is made known and is initiated with the final entryname of the pathname as its reference name. Then this reference name is used along with any entry point name that was given. Since the segment is initiated with a reference name, the user need type only the reference name for subsequent command invocations.

The arguments are character strings that pass information, such as pathnames, access modes, and search strings, to the command from the command processor. An argument may contain white space if the entire string is enclosed by quotation marks (e.g., "Ann Smith"). The order in which arguments are typed is often significant. Arguments beginning with a hyphen (e.g., -brief) are control arguments and specify optional modifications to the operation of the command. Control arguments may also take arguments; it is usually necessary to enclose a control argument string in quotes when it contains white space. Command descriptions contained in the various Multics manuals include descriptions of the specific arguments accepted by the commands.

Multics terminal input allows read-ahead; therefore, the user does not have to wait for a ready message before typing another command line. However, unless the "polite" I/O modes (described in the tty_ I/O module write-up in the Subroutines manual) are specified, the user can be interrupted while typing a line by the ready message or by output printed by the previous command. If an interruption occurs, the line being typed by the user may be hard to read on the terminal. Therefore, the entire line should be killed and retyped (see "Erase and Kill characters" below). The printing of ready messages can be turned off and on using the ready_off and ready_on commands (see the descriptions of these commands in the Commands manual).

Simple Command Line

The general form of a Multics command invocation is:

```
command_name argument1 argument2 ... argumentN
```

where the elements are separated by white space. The rename command, for example, takes arguments in pairs. The first of each pair is the current pathname of the segment to be renamed and the second is the desired new entryname. Thus:

```
rename square_root sqrt
```

causes the command processor to search for and invoke a command called rename at the entry point called rename (rename$rename), with two character string arguments, as the following code fragment represents:

```
x: proc;
declare rename entry options (variable);
call rename ("square_root", "sqrt");
end x;
```

Suppose a user knows that an experimental version of the rename command resides in the directory >Smith_dir. Typing:

```
>Smith_dir>rename square_root sqrt
```

invokes the experimental version instead of the version that would normally have been found by the user's search rules. Subsequent command lines using only the reference name rename continue to invoke the one listed in >Smith_dir.

Compound Command Line

When more than one command invocation appears on one line, a semicolon (;) must be used after each complete invocation except the last one. For example:

```
cwd >old_source; delete program.pl1
```

The space between the semicolon and the next command used in this example is not required.

When the command processor detects an error in the command line, it stops processing at that point. If, for instance, the change_working_dir command in the example above is misspelled as "dcw," the error message "Segment dcw not found" is printed, and the command line is not executed. If a misspelling occurred instead in the delete command (e.g., dlete), the first command would have already been executed, and processing would stop when the misspelled command name was detected.

If, on the other hand, an error is detected by the command program invoked, processing of the remaining commands in the line is completed. For instance, if the pathnname ">old_source" in the example above is misspelled, the error message "change_working_dir: Entry not found" is printed, but the rest of the command line is processed, nevertheless.

Reserved Characters and Quoted Strings

The Multics command language reserves some characters to which special significance is attached. The reserved characters are: space, quotation mark ("), semicolon (;), the newline character, left and right brackets ([and]), left and right parentheses, and the vertical bar (|). Occasionally, however, it is necessary to use a reserved character without its special meaning. The quotation mark character (") is reserved for this purpose. Reserved characters within a quoted string (i.e., a string of characters surrounded by quotation marks) are treated as ordinary characters.

Take for example the case in which a semicolon must not be interpreted as the end of a complete command invocation the first time the line is scanned by the command processor:

```
answer no "dprint ab.list; rename foo ab2.list; dprint ab2.list"
```

Because the answer command provides its answer during execution of only one command line (see the Commands manual), the series of three command lines in this example must be interpreted as one; otherwise the answer command would no longer be in effect when it might be needed, during execution of the rename command. If the rename command then required an answer to execute, it would have to wait for the user to type an answer rather than taking the answer from the answer command as was originally intended. The quotation marks here cause the command processor to ignore the semicolon's normal purpose of marking the end of a command invocation and instead include the semicolons along with the commands they separate as part of the argument to the answer command.

It may also be necessary at times to suppress the special meaning of the quotation mark. For this purpose, doubled quotation marks *within* a quoted string are interpreted as a single quotation mark.

For example, the command line:

```
string  She said, "Hi."
```

would print the string without the quotation marks:

```
She said, Hi.
```

because the command processor strips off single quotation marks, on the assumption that they are present to indicate that the characters within should not be given special interpretation. In order to preserve the quotation marks around the word Hi, the user must add double quotation marks to the quoted string:

```
string  She said, """"Hi."""""
```

Here the command processor will strip off the outer quotation marks and interpret the remaining double marks as single quotation marks:

```
She said, "Hi."
```

Doubled quotation marks that are not enclosed in quotes themselves do not represent a single quotation mark, but the null string. For example:

```
string  She said, ""Hi.""
```

does not result in the string "Hi." but simply the string Hi. Typing the command:

```
string  one ""two""
```

results in the output one two. To obtain the result one "two" the user must type:

```
string  one """two"""
```

or, what might be more likely:

```
string "one ""two"""
```

Iteration

The iteration facility of the command language provides economy of typing for the user who wishes to repeat a command with one or more elements changed. The iteration set consists of two or more elements enclosed by parentheses. Each element of the set, in turn, replaces the entire iteration set in the command line. For example, the command line:

```
print (a b c).pll
```

equivalent to the three commands:

```
print a.pll; print b.pll; print c.pll
```

Iteration can be used with command names as well. The command line:

```
(print delete) test.pll
```

expands into:

```
print test.pll; delete test.pll
```

More than one iteration set can appear in a command. The corresponding element from each set is taken. For instance, the compound command:

```
rename >Smith_dir>(Jones Doe Brown)  (Day White Green)
```

expands into the commands:

```
rename >Smith_dir>Jones Day

rename >Smith_dir>Doe White

rename >Smith_dir>Brown Green
```

Nested iteration sets are also allowed. Evaluation of parentheses occurs from left to right. The principal use of nested iteration sets is to reduce typing when subsets of an element are repeated. For example:

```
create_dir >Smith_dir>(new>(first second) old>third)
```

creates three directories:

```
>Smith_dir>new>first

>Smith_dir>new>second

>Smith_dir>old>third
```

The ability of the Multics command language to concatenate character strings underlies the iteration feature. See "Concatenation" below.

Active Strings

An active string is a substring of a command line, delimited by brackets ([and]). Active strings are evaluated by the command processor and replaced by their values before further processing of the command line occurs. An active function is a program designed to accept character string arguments from the command processor, and return a character string value to the command processor during evaluation of an active string. Standard Multics active functions are described in the Commands manual. To create new active functions, see Section 4 of this manual.

The simplest form of an active string is:

```
[af arg1...argN]
```

where af is the name of an active function and arg_ are character string arguments to the active function.

The command line:

```
send_message [last_message_sender]  Thanks for the link.
```

contains an active string that calls the last_message_sender active function return a value. The command processor returns the value, here expressed in the form "Person_id.Project_id," to the command line. The complete command line is then evaluated and executed, with the returned value of the active function as one element of the command and the message as another.

Active strings can be nested. The following example from a start_up.ec prints a calendar on the first day of each month:

```
if [equal [day] 1] -then calendar
```

Each time the command processor encounters a right bracket (]) it turns to the matching left bracket ([) and evaluates the enclosed active string. This means that nested active strings are evaluated recursively, the innermost string first. To execute the above line, the command processor evaluates [day] and returns, for example, 17, and then evaluates [equal 17 1], which returns false, before it evaluates and executes the complete command line. Expansion of the above example proceeds as follows:

```
if [equal [day] 1] -then calendar
if [equal 17 1] -then calendar
if false -then calendar
```

and the calendar is not printed.

Iteration can be combined with active strings. For example, if the segment fred is a single line consisting of the names of three segments david, robert, and suzanne, then the command line:

```
underline [contents fred]
```

prints out the returned value:

```
david robert suzanne
```

using the contents active function, which returns the contents of the specified segment.

The command line:

```
underline ([contents fred])
```

expands into:

```
underline (david robert suzanne)
```

which then expands into:

```
underline david; underline robert; underline suzanne
```

The command line is then evaluated, and prints:

```
david
robert
suzanne
```

After an active string is evaluated, the return value is rescanned for active strings before being inserted into the command line. For example, if the segment jed is one line consisting of the string ([contents fred]) described above, then the command line:

```
underline [contents jed]
```

invokes [contents fred] as an active function, and the underline command prints:

david
robert
suzanne

The user can suppress rescanning of the returned string for command language special characters by placing a double vertical bar (||) before the active function. The entire return value is then treated as a single token. Using the active functions defined above,

```
underline ||[contents jed]
```

expands only once, and prints:

([contents fred])

where the argument of the underline command is the literal string ([contents fred]).

An active string invocation preceded by a single vertical bar (|) is rescanned only for quotes and white space. For example, the command line:

```
string (|[do "A B]D E"])
```

prints:

```
A
B]D
E
```

When an active function returns two or more elements enclosed by parentheses, rescanning the return value causes iteration to take place in the same manner as that described above. If the segment ned is one line consisting of the parenthetical expression (dave bob sue), then the command line:

```
underline [contents ned]
```

prints:

dave
bob
sue

All of the above examples use active strings consisting of a single active function. In its most general form, an active string can consist of any number of valid active functions separated by semicolons:

```
string [plus 3 4; times 5 6]
```

The value of the active string is then the concatenation of the returned values separated by spaces:

```
7 30
```

Iteration inside an active string has a similar effect; it causes the iterated returned values to be concatenated with intervening spaces. For example, the command line:

```
string [(plus times) 2 3]
```

prints:

```
5 6
```

A vertical bar before a right bracket (|]) eliminates the intervening space when return values are concatenated. For example, the command line:

```
string [plus (1 2) 4 |]
```

prints:

```
56
```

The following example is a more complicated illustration of the use of iteration inside active strings. Assume there are two segments, one.bind and two.bind, in the current working directory. In order to append each segment with the suffix .bind to an archive whose name has the trailing components bound.archive, two active functions are used with the archive command. The strip_entry active function (short name is spe) removes the last component of the given entryname, and the segments active function (short name is segs) returns segment names that match the given star name. The command line:

```
archive a ([spe ([segs *.bind])]).bound ([segs *.bind])
```

expands as follows:

```
archive a ([spe ([segs *.bind])]).bound ([segs *.bind])
archive a ([spe (one.bind two.bind)]).bound ([segs *.bind])
archive a (one two).bound ([segs *.bind])
archive a (one two).bound (one.bind two.bind)
archive a  one.bound one.bind; archive a two.bound two.bind
```

Concatenation

In the Multics command language, the value of a delimited element (a character string bounded by command language reserved characters) in a command line is concatenated with the string or delimited element adjacent to it when no space is placed between the two. The user therefore has the ability to form character strings by concatenation with elements such as parenthetical expressions, active functions, and quoted strings. For example, the home_dir active function turns the character string representation of the pathname of the user's home directory. The user can type a command (presumably from some other directory) such as:

```
rename [home_dir]>square_root sqrt
```

and have the first argument to rename be the concatenation of the value of the home_dir active function with the string >square_root.

Concatenation is permissible in either direction with regard to the delimited string and the nondelimited string. For example, using the active string [contents fred] as described in "Active Strings" above, the command line:

```
delete >project_dir>Doe>([contents fred])
```

deletes the segments david, robert, and suzanne in the directory >project_dir>Doe.

More than one delimited element can be concatenated. For example:

```
delete [home_dir]>([contents fred])
```

deletes the segments david, robert, and suzanne in the user's home directory.

TYPING CONVENTIONS

Three categories of typing conventions are dealt with in this discussion: canonical form, erase and kill characters, and escape characters.

The Multics standard character set is the revised U.S. ASCII Standard (refer to USA Standards Institute, "USA Standard X3.4-1968"). The ASCII set consists of 128 7-bit characters. These are stored internally, right-justified, in four 9-bit fields per word. The two high-order bits in each field are expressly reserved for expansion of the character set; no system program may use them. Any hardware device that is unable to accept or create the full character set should use established escape conventions for representing the set (see "Escape Characters" below). There are no meaningful subsets of the revised ASCII character set.

The ASCII character set includes 94 printing graphics, 33 control characters, and the space. Multics conventions assign precise interpretations to all the graphics, the space, and 10 of the control characters. The remaining 23 control characters are presently reserved. The full ASCII character set appears in Appendix A.

Canonical Form

A character stream is a representation of one or more printed lines. Since the same printed line can be produced using different sets of keystrokes, there are several possible character streams that represent the same line. For example, the line:

```
start    lda  alpha,4     get first result.
```

could have been typed with either spaces or horizontal tabs separating the fields; one cannot tell by looking at the printed image.

A program should be able to compare two character streams easily to see if they produce the same printed image. It follows that all character input to Multics must be converted into a standard (canonical) form. Similarly, all programs producing character output, including editors, must produce canonical form output streams.

Of all possible ASCII character strings, only certain strings are ever found within Multics. All strings that produce the equivalent printed effect on a terminal are represented within Multics as one string, the canonical form for the printed image. The user, however, is free to type a noncanonical character stream. This stream is automatically converted to the canonical form before it reaches his program. An exception to this automatic conversion is that tab characters are preserved; a detailed description of the conversion process is found later in this section. If the user wants his program to receive raw or partially processed input from his terminal, an escape mechanism is provided by the modes operation of the tty_ I/O module. The I/O module is accessed via a call to the iox_ subroutine (see the description of the iox_ subroutine in the Subroutines manual). The modes available that apply to canonicalization are:

`^can`
: no canonicalization of overstrikes.

`^esc`
: no canonicalization of escape characters.

`^erkl`
: no erase and kill processing.

`rawi`
: read the specified data from the terminal without any conversion or processing. This includes shift characters and undifferentiated uppercase and lowercase characters.

Similarly, an I/O module is free to rework a canonical stream on output into a different form if, for example, the different form happens to print more rapidly or reliably on the device.

The current Multics canonical form is designed for the convenient typing of aligned tabular information, which requires an ambiguous interpretation of the tab character. The following three statements describe the current Multics canonical form.

1. A text line is a sequence of character positions separated by horizontal carriage motion and ending in a newline character.

2. Carriage motion consists of newline, tab, and space characters.

3. A character position consists of a single graphic or several overstruck graphics. A graphic is a printable character. An overstruck character position consists of two or more graphics separated by backspaces. Regardless of the order in which the graphics are typed, they are always stored in ascending ASCII order. Therefore, the symbol "<>_", whether typed as:

   ```
   >B<B_
   ```

   or

   ```
   <B>B_
   ```

   or

   ```
   _B<B>
   ```

   is always stored internally as:

   ```
   <B>B_
   ```

   where B is a backspace.

There are any number of ways to type two or more consecutive overstruck character positions. The graphics in each position are grouped together, so that "x̲x̲" is always stored as:

```
<B>B_<B>B_
```

The following paragraphs give a complete set of rules for transforming a typed line into the form in which it is stored, followed by further examples illustrating the rules. The transformation process is carried out in three steps: canonicalization, erase/kill processing, and escape processing. If two or more of the rules listed below are applicable to a given input string, they are applied in the order in which they are presented here.

Canonicalization

Canonicalization is the process of converting an input string into canonical form. Two methods of canonicalization are defined on Multics: a method for printing terminals and a method for video (CRT) terminals. Both methods of canonicalization attempt to ensure that what is visible on the terminal is the canonical form of the input string. The method used is determined by the setting of the "can_type" mode, as explained in the description of the tty_ I/O module (see the Subroutines manual).

Canonicalization for printing terminals (overstrike canonicalization) is designed for terminals which are capable of overstriking multiple characters in a single column. Any group of overstruck characters is converted to a single representation regardless of the order in which the characters were entered into the column.

Canonicalization for video terminals (replacement canonicalization) is designed for terminals which are not capable of overstriking. When a character is entered into a column, any characters previously present in that column are no longer visible. Replacement canonicalization mimics this behavior of the terminal by only placing the last character typed into any column into the canonical representation of the string.

The canonicalization process consists of two distinct steps: column assignment, which is identical for both methods of canonicalization, and the actual canonicalization process.

Column Assignment

The following rules are used to determine which printing graphics, if any, appear in each physical column position.

1. The leftmost position of the carriage is considered to be column 1.

2. Each printing graphic or space typed increases the column position by 1.

3. Each backspace typed decreases the column position by 1 unless the column position is 1.

4. A carriage return sets the column position to 1.

5. A horizontal tab increases the column position to the next tab stop; tab stops are defined to be at columns 11, 21, 31, etc.

6. A newline, formfeed, or vertical tab sets the column position to 1 and advances the carriage vertically; thus no character typed after such a character can share a column position with a character typed before it.

7. If the terminal is not in ctl_char mode, any ASCII control character other than backspace, horizontal tab, newline, vertical tab, formfeed, and carriage return is discarded. If the terminal is in ctl_char mode, such characters are treated as if they were printing graphics (with the exception of the NUL character, which is always discarded). The default is that ctl_char mode is off.

Overstrike Canonicalization

The following rules determine the formation of the canonical string.

1. Characters on each line are sorted so that their associated column positions are monotonically increasing.

2. No carriage return characters may appear in the canonical string.

3. A horizontal tab is preserved as typed unless a printing graphic appears in one of the columns skipped by the tab, in which case the tab is replaced by an appropriate number of spaces.

4. Backspaces appear in the canonical string only when two or more printing graphics share a column position.

5. When two or more different printing graphics share a column position, the characters are sorted as follows: graphic with lowest numeric ASCII code, backspace, graphic with next lowest numeric ASCII code, etc.

6. If the contents of a column position consist of two or more instances of the same printing graphic, that column is reduced to a single instance of the graphic.

7. A line-ending character (newline, formfeed, or vertical tab) immediately follows the last printing graphic in the rightmost column position on the line.

Overstrike Canonicalization Examples

Several illustrations of canonical form are shown below. Assume that the typist's terminal has horizontal tab stops set at 11, 21, 31, etc.

```
Typist:          this is ordinary text.N
Typed line:      this is ordinary text.
Canonical form: this is ordinary text.N
```

where N is the newline character. In most cases, the canonical form is the same as the original key strokes of the typist, as above.

```
Typist:          here fullBBBB___ means thatN
Typed line:      here full means that
Canonical form: here _Bf_Bu_Bl_Bl means thatN
```

where B is a backspace and N is a newline character. This is the most common type of canonical conversion, done to ensure that overstruck graphics are stored in a standard pattern.

```
Typist:          We see no probSBlemC__N
Typed line:      We see no problem
Canonical form: WB__Be see no problemN
```

where B is a backspace, N is a newline character, S is a space, and C is a carriage return. The space between "prob" and "lem" was not overstruck; it and the following backspace were simply removed. Note the difference in the storage of the characters that *were* overstruck in this and the preceding example; the ASCII code value of the underscore is between the values for uppercase and lowercase letters.

Replacement Canonicalization

Replacement canonicalization is designed for use on a terminal with the following characteristics:

- Overstriking a character with any other printing character or a space causes the first character to be erased.

- Entering a tab character simply moves the cursor position to the next tab stop (column 11, 21, etc.) without erasing any intervening characters.

The following rules determine the formation of the canonical string:

1. Characters on each line are sorted so that their associated column positions are monotonically increasing.

2. No carriage return characters may appear in the canonical string.

3. A horizontal tab is preserved as typed unhess a printing graphic appears in one of the columns skipped by the tab, in which case the tab is replaced by an appropriate number of spaces.

4. When two or more characters (including space and identical printing graphics) share a column position, the last character entered by the user in that column is kept and all other characters in that column discarded.

5. A line-ending character (newline, formfeed, or vertical tab) immediately follows the last printing graphic in the rightmost column position on the line.

6. When in ctl_char mode, a control character (other than the carriage motion characters HT, BS, CR, NL, VT, and FF) shares the column position of the immediately following graphic or space. If the control character is followed by a horizontal tab, it shares the first column skipped over by the tab. Such a control character is not affected by backspace (i.e., it is not removed if the graphic sharing its column position is replaced).

With replacement canonicalization, as seen above, it is not possible to overstrike two characters, as the last one typed is always the only character in that column. Thus it is not possible to use the feature of overstriking a character with the erase character, as described in the "Erase and Kill Characters" section following, to delete a character typed in the middle of a line. Instead, to delete such a character, you must reposition to the character in question and retype the remainder of the line being input.

Therefore, you may want to disable the erase character when using replacement canonicalization. This may be accomplished by the command line:

```
set_tty -edit \400
```

where `\400` is a character which cannot normally be entered on the terminal.

Replacement Canonicalization Examples

Several illustrations of canonical form are shown below. Assume that the typist's terminal has horizontal tab stops set at 11, 21, 31, etc.

```
Typist:          this is ordinary text.N
Screen contents: this is ordinary text.
Canonical form:  this is ordinary text.N
```

where N is the newline character. In most cases, the canonical form is the same as the original key strokes of the typist, as above.

```
Typist:          this is a msitake.BBBBBBBisN
Screen contents: this is a mistake.
Canonical form:  this is a mistake.N
```

where B is a backspace and N is a newline character. This example illustrates the correction of errors in the middle of a typed line. It is the most common use of replacement canonicalization.

```
Typist:          this si a strange BBBBBBBBBBBBisHHBBexample.N
Screen contents: this is a strange example.
Canonical form:  this is a strange example.N
```

where B is a backspace, H is a horizontal tab, and N is a newline character. This example illustrates that the horizontal tab character does not erase intervening characters (" a strange" in this example).

```
Typist:          This is   some text.BBBBBBBBBBBBsome text.   N
Screen contents: This is some text.
Canonical form:  This is some text.N
```

where B is a backspace and N is a newline character. This example illustrates that in order to erase extra whitespace in a line, the typist must position to the first extraneous character, retype the remainder of the line, and type sufficient spaces at the end of the line to overstrike any extra undesired characters.

If, in the above example, the final spaces are not typed, the following occurs:

```
Typist:          This is   some text.BBBBBBBBBBBsome text.N
Screen contents: This is some text.t.
Canonical form:  This is some text.t.N

Typist:          use of cXontrol charactersN
Screen contents: use of control characters
Canonical form:  use of cXontrol charactersN
```

where N is a newline, X is any non-carriage-motion control character, and the terminal is in ctl_char mode.

```
Typist:          Don't remove the vpXmBBBcontrol character.N
Screen contents: Don't remove the control character.
Canonical form:  Don't remove the coXntrol character.N
```

where B is a backspace, N is a newline, X is any non-carriage-motion control character, and the terminal is in ctl_char mode. Note that the control character remains in the column position at which it was entered, even though the graphic in that position has been replaced.

Erase and Kill Characters

Two capabilities for minimally editing the line being typed are available. They are:

- The ability to delete the latest character or characters (erase)
- The ability to delete all of the current line (kill)

By applying canonical form to these two editing functions, it is possible to interpret unambiguously a typed line in which editing was required.

The first editing convention reserves one graphic as the erase character. On Multics, the default erase character is the number sign (#). The user can designate a different character by invoking the set_tty command with the -edit control argument. Although the erase character is a printed graphic, it does not become part of the line. When it is the only graphic in a print position, it erases itself and the contents of the previous print position. Several successive erase characters erase an equal number of print positions. One erase character typed immediately after "white space" causes the entire white space to be erased (any combination of tabs and spaces is called white space). The number sign can be struck over another graphic. In this case it erases the print position on which it appears. For example, typing:

```
     theSSne###next
or
     theST#next
or
     the#next
```

where S is a space and T is a horizontal tab, produces:

```
     thenext
```

Since processing of erase characters takes place after the transformation to canonical form, there is no ambiguity as to which graphic character has been erased. The printed image is always the guide.

The second editing convention reserves another graphic as the kill character. On Multics, the default kill character is the commercial at sign (@). Again, the user can redesignate this. When this character is the only graphic in a print position, the contents of that line up to and including the kill character are discarded. Again, since kill processing occurs after the conversion to canonical form, there is no ambiguity about which characters have been discarded.

By convention, an overstruck erase character is processed before a kill character, and a kill character is processed before a nonoverstruck erase character. Therefore, the only way to erase a kill character is to overstrike it with an erase character.

Because of their special meanings to Multics, these two graphics should be avoided in software.

The following rules apply to erase and kill characters.

1. If the terminal is in esc mode, an erase or kill character alone in a column immediately preceded by an escape character alone in a column is not processed as an erase or kill character.

2. An erase character alone in a column position and preceded by more than one blank column results in the deletion of *all* immediately preceding blank columns, as well as of the erase character.

3. An erase character alone in a column position results in the deletion of itself and of the contents of the preceding column position.

4. An erase character sharing a column position with one or more printing graphics results in the deletion of the contents of that column position.

5. A kill character results in the deletion of its own column position and all column positions to its left, unhess it shares a column position with an erase character, in which case rule 4 applies (the kill character is erased).

Notice that for rule number 1 to apply, the erase or kill character must actually have been typed in the column immediately following the escape character. The reason for this is that it facilitates the erasing of escape sequences, e.g., \001####.

Examples of Erase and Kill Processing

```
Typist:          abcx#deSBfzz##gN
Typed line:      abcx#defzz##g
Canonical form:  abcx#defzz##gN
Final input:     abcdefgN


Typist:          this@In the offBBB___##nB_ stateN
Typed line:      this@In the off##n state
Canonical form:  In the _Bo_Bn stateN
Final input:     In the on state
```

Escape Sequences

Some terminals cannot print all 128 ASCII characters. To maintain generality and flexibility, standard software escape conventions are used for all terminals. Each class of terminal has a particular character assigned to be the software escape sequence character in the terminal type file. When this character occurs in an input (or output) string to (or from) a terminal, the next character (or characters) are interpreted according to the conventions described below. The escape sequence character should not be confused with the ASCII ESC, which is octal 033.

The standard escape sequence character in Multics is the left slant (\); like the erase and kill characters, it should be avoided in Multics software. The universal escape conventions are:

1. The string \d1d2d3 represents the octal code d1 d2 d3 where di is a digit from zero to seven. Any arbitrary character can be represented this way. The string \d2d3 is equivalent to \d1d2d3 if d1 is zero. The string \d3 is equivalent to \d1d2d3 if d1 and d2 are zero.

2. Local (i.e., concealed) use of the newline character that does not go into the computer-stored string on input, and is not in the computer-stored string on output, is effected by typing \<newline character>.

3. The characters \# place an erase character into the input string.

4. The characters \@ place a kill character into the input string.

5. The characters \\ place a left slant character into the input string.

The escape conventions described in items 1 through 5 above apply only if none of the characters involved are overstruck.

The following rules apply to escape sequences.

1. An escape sequence consists of an escape sequence character alone in its column position followed by one or more printing graphics each of which is alone in its column position. An escape sequence is replaced by a single character in the canonical string.

2. An escape sequence consisting of two successive escape sequence characters is replaced by an escape sequence character.

3. An escape sequence consisting of an escape sequence character followed by an erase or kill character is replaced by an erase or kill character.

4. An escape sequence consisting of an escape sequence character followed by one, two, or three octal digits is replaced by the character whose ASCII value is represented by the sequence of octal digits.

5. An escape sequence character followed by a newline character results in the deletion of both characters from the canonical string.

6. Other escape sequences may be defined on a per-terminal-type basis, where such a sequence consists of an escape sequence character and one character following.

7. If the character following an escape sequence character does not result in an escape sequence as defined by the six rules above, the escape sequence character and following characters are stored as they appear on the line.

TYPING CONVENTION EXAMPLES

In the examples below, the following conventions are used:

N	represents a newline
C	represents a carriage return, assuming that the mode lfecho is not set
B	represents a backspace
T	represents a horizontal tab
S	represents a space
{nnn}	represents a character whose ASCII value is nnn (octal)
\	is the escape sequence character
#	is the erase character
@	is the kill character

The examples in the first group illustrate how various typed sequences are canonicalized in terms of column position; these are followed by examples of erase, kill, and escape canonicalization. In the second group, lines are shown as they appear physically, with no consideration given to the precise sequence of keystrokes that might have produced them.

Column Canonicalization Examples

```
Typed:      nothing special about this line.N
Appearance: nothing special about this line.
Result:     nothing special about this line.N


Typed:      extraneous white sSBpace is ignored.CSN
Appearance: extraneous white space is ignored.
Result:     extraneous white space is ignored.N


Typed:      Here are two ways (2B_) to overstrike.C____N
Appearance: Here are two ways (2) to overstrike.
Result:     HB__Be_Br_Be are two ways (2B_) to overstrike.N


Typed:      tab + backspace isTBreduced to spaces.N
Appearance: tab + backspace is    reduced to spaces.
Result:     tab + backspace isSSSSreduced to spaces.N
```

NOTE: See rule 3 under "Formation of the Canonical String" above.

Erase, Kill, and Escape Examples

The first few examples illustrate erase and kill processing; the remaining examples illustrate both escape processing and erase and kill processing. These examples assume the terminal is in esc mode (mentioned in rule 1 under "Erase and Kill Characters" and described in the tty_ I/O module) and that overstrike canonicalization is being used.

```
Typed:      abz#cde
Appearance: abz#cde
Result:     abcde


Typed:      abSSS#cde
Appearance: ab   #cde
Result:     abcde
```

```
Typed:      not@neverSobB#nSMonday
Appearance: not@never o#n Monday.
Result:     never on Monday.


Typed:      nox#wBBBBB___S_Sit'sSright.
Appearance: nox#w it's right.
Result:     now it's right.


Typed:      noxBBB____B#wB_Sit'sSright.
Appearance: nox#w it's right.
Result:     noxw it's right.
```

NOTE: Erase character is overstruck; see rule 4 under "Erase and Kill Characters" above.

```
Typed:      dclSrrsScharS(1)SstaticSinit("\017#6");
Appearance: dcl rrs char (1) static init("\017#6");
Result:     dcl rrs char (1) static init("{016}");


Typed:      \023B_
Appearance: \023
Result:     {002}3
```

NOTE: Overstruck 3 is not part of escape sequence.

```
Typed:      \B_112
Appearance: \112
Result:     \112
```

NOTE: Overstruck \ is not an escape character.

```
Typed:      a\##b
Appearance: a\##b
Result:     a\b
```

NOTE: According to rule 1 under "Erase and Kill Characters," the first # is not an erase character; according to rule 3 under "Erase and Kill Characters," the second # erases itself and the preceding #.

```
Typed:      a\@#b
Appearance: a\@#b
Result:     a\b
```

NOTE: Same note as in immediately preceding example.

```
Typed:      a\B#@b
Appearance: a¥@b
Result:     b
```

NOTE: The \ is erased by the overstruck #.

```
Typed:      a\\#b
Appearance: a\\#b
Result:     a\#b
```

NOTE: According to rule 1 under "Erase and Kill Characters," erase canonicalization does not recognize the #; according to rule 2 under "Escape Sequences," escape canonicalization recognizes \\ and attaches no special meaning to the #.

```
Typed:      a\\##b
Appearance: a\\##b
Result:     a\b
```

NOTE: According to rules 1 and 3 respectively under "Erase and Kill Characters," the first # is not an erase character and the second # erases itself and the preceding #; according to rule 2 under "Escape Sequences," \\ reduces to \.

```
Typed:      a\\###b
Appearance: a\\###b
Result:     a\b
```

NOTE: The first # is not an erase character; the next two are, erasing the second \ and the first #.

```
Typed:      a\\####b
Appearance: a\\####b
Result:     ab
```

NOTE: The first # is not an erase character, and must be erased before the two \ characters. The previous examples illustrate the difficulty of erasing a double \; the clearest method is probably to overstrike (a¥¥b) .

```
Typed:      a¢<#b (typed on an IBM Model 2741-like terminal)
Appearance: a¢<#b
Result:     a\b
```

NOTE: Only the < is erased; ¢ is translated to \ (see "Escape Conventions on Various Terminals" below).

TERMINAL OUTPUT

Certain transformations are performed on output destined for a terminal to ensure that it is displayed correctly. These transformations can be broken down into the following categories: carriage motion, delays, escape sequences, continuation lines, and end-of-page processing.

Carriage Motion

Six entries in the terminal's special characters table specify the character sequences to be output when any of the various carriage motion (space, formfeed, vertical tab, horizontal tab, backspace, carriage return, and newline) characters are encountered (for information on this table, see the description of the set_special order to the tty_ I/O module in the Subroutines manual). The most usual case is that the sequence for newline consists of carriage return followed by newline (i.e., linefeed), and each of the other sequences either consists of the source character itself or is null to indicate that the specified function is not available.

In general, carriage motion is reduced to its simplest and most efficient form. Any combination of consecutive carriage motion characters is output as net right or left motion, e.g.,:

```
SSBSS
```

is output as:

```
SSS
```

where S is a space and B is a backspace. If a newline immediately follows other carriage motion characters, those carriage motion characters are omitted. In addition, a combination of spaces and horizontal tabs that moves the carriage to or over a tab stop is converted to tabs followed by the minimum possible number of spaces. Tab stops are assumed to be at columns 11, 21, 31, etc. Thus the following sequence (starting at column 1):

```
abcdSSSSSSSSSef
```

is converted to:

```
abcdTSSSef
```

where S is a space and T is a horizontal tab. An exception arises if the terminal is in ^tabs mode or if the special characters table specifies a zero-length sequence for horizontal tabs. In either of these cases, all rightward carriage motion is output as spaces; as many spaces are output as necessary to reach the appropriate column position.

Net left carriage motion is normally output as backspaces unless the final column position is so near the left margin that it is more efficient to output a carriage return followed by spaces. Thus:

```
abcdefgCSSSS___
```

is output as:

```
abcdefgBBB___
```

where as:

```
abcdefghijkBBBBBBBBB__
```

is output as:

```
abcdefghijkCSS__
```

where C is a carriage return, S is a space, and B is a backspace.

If the terminal lacks the capability to perform a carriage return without a linefeed, the carriage return sequence in the special characters table should be null, in which case net left carriage motion is always output as backspaces. Conversely, if the terminal lacks the backspace function, the backspace sequence should be null, and all net left carriage motion is output as a carriage return followed by spaces. If both sequences are null, net left carriage motion is ignored.

Delays

Printing terminals frequently require more than one character time to move the carriage in any way other than one position to the right. In order to allow the terminal time to reach the column position in which it is next supposed to print, the Multics Communication System may output one or more ASCII NUL characters following a carriage motion character. NUL characters used in this way are called delays.

The number of delays required in any given situation depends on the terminal mechanism, the distance the carriage has to travel, and the speed at which characters are sent to the terminal (baud rate). The delay table (described under the set_delay order to the tty_ I/O module in the Subroutines manual) contains values, appropriate to the particular terminal and baud rate, that determine the number of delays required for any carriage motion character causing a number of columns to be traversed. The terminal type file (TTF), described in Appendix B, contains a specification of delay tables to be used at various speeds for each terminal type. To construct a new terminal type entry, it may be necessary to obtain formulas from the terminal manufacturer from which the necessary delay table values can be derived.

Output Escape Sequences

A character that a particular terminal is incapable of printing may be represented by an escape sequence. The substitution of an escape sequence for a particular character is dictated by that character's entry in the output conversion table (described under the set_output_conversion order to the tty_ I/O module). Two kinds of escape sequences are defined: octal escape sequences, and special escape sequences. An octal escape sequence, as explained earlier, consists of a left slant character followed by three octal digits representing the ASCII value of the character being replaced (e.g.,\012). A special escape sequence is one specified in the special characters table, and consists of zero to three arbitrary characters. Each special escape sequence has two forms, one used in edited mode and one used in ^edited mode. See the descriptions in the Subroutines manual of the set_output_conversion order, the set_special order, and edited mode for the tty_ I/O module for more detailed information.

Continuation Lines

When the length of an output line (i.e., the number of column positions between two newline characters) exceeds the terminal's physical paper or screen width, a newline sequence is inserted and the excess characters appear on the following line, preceded by a continuation sequence consisting of the characters \c. A "line" of arbitrary length can be output using as many continuation lines as necessary. The physical line length of the terminal is made available to the software by means of the line length (ll) mode of the tty_ I/O module.

End-of-Page Processing

The page length (pl) mode of the tty_ I/O module may be used to specify the physical length in lines of a page. This feature is primarily of interest to users of video display terminals as a means of preventing output from being scrolled off the screen before it can be read. If page-length checking is enabled, then the last line of a page contains a warning string consisting of the end-of-page sequence specified in the output conversion table (described under the set_output_conversion order to the tty_ I/O module in the Subroutines manual); this sequence is normally the characters "EOP". The output stops when the page is full, and restarts when the user types a newline or formfeed character. If the end-of-page sequence is a null string, output stops at the right margin of the last line of the page, and no warning string is displayed. See the descriptions of pl and scroll modes for further information.

ESCAPE CONVENTIONS ON VARIOUS TERMINALS

The following paragraphs list escape conventions for some of the terminals that can be used to access the Multics system. In general, the conventions described here apply to logging in and out as well as to all other typing. For user convenience, terminals should support the full (128 characters) ASCII character set on input and output. For terminals that do not have a full ASCII character set, escape conventions have been provided. Any of these escape conventions, however, can be respecified by the user.

Upper-Case-Only Devices

Because these models do not have both uppercase and lowercase characters, the following typing conventions are necessary to enable users to input the full ASCII character set:

1. The keys for letters A through Z input lowercase letters a through z, unless preceded by the escape character \ (left slant). The left slant is shift-L on the keyboard, although it does not show on all keyboards. For example, to input "Smith.ABC", type "\SMITH.\A\B\C".

2. Numbers and punctuation marks map into themselves whenever possible. The underscore (_) is represented by the back arrow (<-). The circumflex (^) is represented by the up arrow (↑). The acute accent (') is represented by the apostrophe (').

3. The following other correspondences exist:

Character	type in	octal
backspace	\-	010
grave accent ( )	\'	140
left brace ({)	\(	173
vertical line (|)	\!	174
right brace (})	\)	175
tilde (~)	\=	176

Execuport 300

The following non-ASCII graphics are considered to be stylized versions of ASCII characters:

back arrow (<-) for underscore (_)

CDI Model 1030

The following non-ASCII graphics are considered to be stylized versions of the ASCII characters:

back arrow (<-) for underscore (_)

up arrow (↑) for circumflex (^)

FLOW CONTROL

Some asynchronous terminals implement a flow control protocol for input and/or output. The following paragraphs describe briefly the mechanisms supported by the Multics system.

Input Flow Control

For terminals that can be used to send high-speed input using a paper tape or cassette tape reader, it is useful for the system to be able to instruct the terminal to stop and start transmission so that the input does not arrive faster than it can be processed. Such terminals (for example the Tektronix 4051) suspend transmission on receipt of a particular character (called the input_suspend character), and resume it on receipt of another character (the input_resume character). In addition, such terminals sometimes suspend input at the end of each tape record or block, possibly transmitting the input_suspend character before doing so. It is the responsibility of the system in this case to request the resumption of input by sending the input_resume character. The input_suspend and input_resume characters may be specified in the description of the terminal type as described in Appendix B, or by means of the input_flow_control_chars order to the tty_ I/O module, described in the Subroutines manual. The timeout option is used to specify that the terminal suspends input without transmitting an input_suspend character, and that the system must send an input_resume character when it detects that input has been suspended. Input flow control is enabled and disabled by means of the iflow mode of the tty_ I/O module.

Output Flow Control

Output flow control is intended to manage terminals that buffer output, since they print or display at less than channel speed. Two types of output flow control protocols are supported by the Multics system. The first, called suspend/resume, is used by various terminals including several made by Digital Equipment Corporation. In this protocol, the terminal sends a particular character (called the output_suspend character) when its buffer is nearly full in order to request that the system temporarily stop sending output. When it is ready to accept more output it sends another character (the output_resume character). The other protocol, called block acknowledgement, is used by various terminals, including the Diablo 1620. In this protocol, the system is expected to subdivide output into blocks no larger than the terminal's buffer, and end each block with a specific character (the end_of_block character). When the terminal is ready to accept more output, it transmits an acknowledgement character. The type of protocol and the specific characters to be used can be specified in the terminal type description as described in Appendix B, or by use of the output_flow_control_chars order to the tty_ I/O module, described in the Subroutines manual. Output flow control is enabled and disabled by means of the oflow mode of the tty_ I/O module.

Hardware Flow Control Using the CTS Dataset Lead

CTS flow control protocol utilizes the capabilities of the FNP's asynchronous communications adaptor to remove the need for delay calculation to manage output flow control and also eliminates the need for output flow control information embedded in the data stream. This flow control is implemented for hardwired asynchronous communications lines (lines utilizing the FNP module 'control_tables'). It provides a stop-on-character output flow control.

This protocol utilizes the CTS dataset lead (pin 5) to control output from the FNP to the Data Termination Equipment (DTE). If CTS is high, output will be sent; when CTS drops, the current character will be finished and output will cease until CTS is raised again.

This protocol is implemented such that a line must have all three leads (CTS, CD and DSR) high to be initially on-line. After this point CTS will act as a flow control signal, until the line is hungup again by dropping either CD or DSR.

Many terminals can utilize this protocol or a DTR flow control protocol. The DTR protocol uses the DTR lead from the terminal (pin 20) in the same manner. For DTR flow control a connector must be wired which connects the terminal DTR to the FNP CTS.

The use of a hardware protocol removes computational loading from the mainframe MCS since delays do not need to be calculated. It also lightens buffer loadings since buffer space for delay characters is not needed.

BLOCK TRANSFER

Some asynchronous terminals are capable of operating in "block mode", i.e., they can be made to buffer a block of data and then transmit it at channel speed in response to a single keystroke. The system may not handle such high-speed input correctly unless it is informed that the terminal is capable of such transmission. The blk_xfer mode of the tty_ I/O module is used for this purpose.

A terminal is suitable for use in blk_xfer mode if it delimits the block or "frame" of data transferred by appending a specified character (the "frame_end" character) to the block and optionally preceding the block with a "frame_begin" character (which need not be different from the frame_end character). The particular characters used will depend on the terminal. The characters used can be specified by the framing_chars statement in the terminal type definition as described in Appendix B, or by means of the set_framing_chars order to the tty_ I/O module.

If the terminal is in blk_xfer mode, and frame_begin and frame_end characters have been specified, all characters starting with a frame_begin character, up to and including the next following frame_end character, are treated as a frame. If a frame_end character has been specified, but no frame_begin character has been specified, then all characters between one frame_end character and the next are treated as a frame. In general, none of the characters in a frame are delivered to the user's process until the end of the frame has been reached. Calls to iox_$get_line still read input one line at a time, but the first line in a frame is not available for reading until the entire frame has been received.

SECTION 4

MULTICS PROGRAMMING ENVIRONMENT

The Multics programming environment is supported by an elaborate set of system procedures and data structures that are generally invisible to the programmer but that greatly affect the ways in which programs are written. For example, because of the Multics virtual memory scheme, a procedure can freely reference any segment in the storage system (to which it has access privileges) without knowing either its size or its physical location. Because the normal mode of program execution uses a stack, most procedures are potentially recursive, even when written in a programming language that does not support recursion. While the supported programming languages provide standard interfaces to the system environment, the programmer is free to use features of the environment in his own way.

The information in this section presents the basic aspects of the programming environment. Program preparation presents the steps involved in implementing a program to run on Multics. Then the section presents the major internal interfaces between a user program and the system that are automatically or explicitly activated during program execution. The remainder of the section is devoted to subsystem writing, including closed and interactive subsystems.

PROGRAM PREPARATION

The basic steps involved in preparing a program to run in the Multics environment and the system features available to perform them are presented below. Specific conventions associated with a particular programming language are described in the appropriate language manual. The end product of the steps described is an object segment constructed to interface with Multics facilities and other object segments. Some of these facilities, such as dynamic linking and process-related data structures, are presented later in this section.

Programming Languages

The major programming languages currently available on Multics are:

PL/I	superset of the American National Standard programming language PL/I, ANSI X3.53-1976. It also conforms to International Standards Organization standard 6160-1979
FORTRAN	superset of ANSI FORTRAN, ANSI X3.9-1966, and all features of ANSI X3.9-1978
COBOL	subset of the ANSI standard COBOL, ANSI X3.23-1974

BASIC	compatible with the Dartmouth Version 6 BASIC and very similar to the ANSI Standard for Minimal BASIC (ANSI X3.60-1978)
ALM	Multics assembly language
APL	interactive interpreter (based on IBM APL)
PASCAL	based on the standard ISO Pascal

Each Multics translator can be called as a command and produces executable object code segments. Such segments can be executed as subroutines or at command level. For information on designing programs compatible with the Multics command environment, see "Writing a Command" and "Writing an Active Function" below.

PL/I is the standard language on Multics (the system itself is written largely in PL/I). Thus, the system is documented in terms of PL/I calling sequences, argument declarations, and standard data types. Areas of the system requiring the use of special hardware instructions are written in ALM.

A program written in any of the Multics programming languages can call other programs written in the same language by merely following that language's calling conventions. And because programs written in different languages produce compatible object segments, programs written in any Multics programming language can call programs written in any other Multics programming language as long as the data types of any arguments passed are recognized data types in both languages. In some cases it may be necessary to create a PL/I interface procedure to handle transmission of arguments between such programs. Individual language descriptions explain restrictions on calls to programs produced by different translators and suggest possible interface mechanisms.

Creating and Editing the Source Segment

A source program resides in an online segment of the Multics storage system. It is initially created and subsequently modified using one of the Multics text editors, such as Emacs, Edm or Qedx. (See the Commands manual for specific descriptions of these text editing facilities.)

The name given a source segment must have the form:

```
source_name.language_name
```

where:

1. source_name
   is the name of the user program.

2. language_name
   is the name of the programming language in which it is written.

Some sample source segment names are:

```
square_root.pl1
square_root.fortran
square_root.basic
```

and the object segments produced from each of these are named square_root.

Creating an Object Segment

To translate a source program into object code, the user issues a command to the appropriate language translator, supplying the source program name as an argument. To compile the source segment named square_root.pl1, the user issues the command:

```
pl1 square_root
```

and an object segment named square_root is produced and placed in the user's working directory.

The user could as well have typed:

```
pl1 square_root.pl1
```

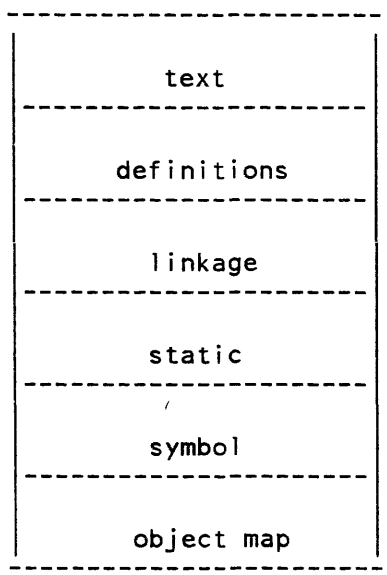
but as the pl1 compiler is defined to operate only on segments ending in the suffix ".pl1", the compiler is allowed to infer the existence of this suffix, provided it actually exists on the segment's name, even if it is not specified in the command line.

Unless the user selects optional features, the only output produced by the translator is the object segment itself and messages describing any errors detected during translation. The user corrects errors by editing the original source segment. Object segments produced by different translators are compatible, although, as stated previously, the difference in data types and representations may require the construction of interfaces to pass arguments among programs written in different languages.

The optional control arguments used by the language translators are also standardized. They provide additional output such as program listings and object maps. When a listing is requested, it is placed in the user's working directory with the name source_name.list (e.g., square_root.list). Of particular interest to users of high-level programming languages is the default control argument -table, which causes a symbol table to be placed in the object segment, thereby enabling the program to be debugged symbolically. (See "Debugging Facilities" below.) When a program is thoroughly debugged, it should be recompiled with the -no_table control argument.

Object-Segment Format

All Multics translators produce a standard object segment that contains object code, linkage data, and other information that may be required at execution time. The overall format of an object segment is shown below.



where:

1. text
   contains the object code, a binary machine-language program.

2. definitions
   contains a set of locations within the segment that can be referenced by name (entry points) and a list of references made by the program to external segments (in character-string form).

3. static
   is a prototype static section, containing PL/I internal static variables. It is usually contained within the linkage section rather than being a separate section. It is copied into the per-process user area when the object segment is first referenced.

4. linkage
   is a prototype linkage section, containing dynamic linkage information. It is copied into the per-process user area when the object segment is first referenced and contains information used by the dynamic linker to resolve external references.

5. symbol
   contains relocation bits for the text and linkage areas (used for binding) and additional information that may be generated by translation options, such as a symbol table.

6. object map
contains lengths and offsets for each section of the object segment.

For a detailed description of an object segment's format and contents, see Appendix G.

Debugging Facilities

Multics provides extensive interactive program debugging facilities through the two commands, probe and debug, which make use of the symbol table placed in the object segment of compile time. The two commands provide similar services, but the debug command is oriented more toward the needs of a machine language programmer while probe is designed with the high-level language programmer in mind. Multics also provides a trace command that traces the flow of control through program execution and a trace_stack command that traces the list of programs active on the program stack. (The debug, probe, trace, and trace_stack commands are described in the Commands manual.)

A central feature of both debug and probe is the facility for setting breakpoints at specified program locations. The program is then executed. When a preset breakpoint is reached, execution is interrupted and the current state of variables preserved. The user can then perform other debugging operations such as examining the values of data items, inserting test values, executing other programs, and so on. Execution can then be continued from the point at which it was suspended.

Writing a Command

Any of the standard Multics compilers can be used to create a Multics command procedure. A command procedure differs from other procedures in the following ways:

1. A command procedure is called by the Multics command processor. Since the input to the command processor is limited to the characters that the user types in the command line, the command processor can only pass nonvarying, unaligned character-string arguments to the command procedure. This means that the command procedure may have to convert these character strings to another data type more appropriate to its needs.

2. A command procedure can receive only input arguments. An error may occur if the procedure changes the value of any of its arguments. Also, the command procedure may not set one of the arguments to indicate the success or failure of its operation.

3. A command procedure must be prepared to handle a variable number of arguments. Many command procedures accept optional control arguments, which may or may not be present. Even command procedures expecting a fixed number of arguments must be prepared to diagnose an error when the user mistakenly types too many or too few arguments.

The command processor provides an environment that supports the differences between command procedures and other procedures. A command procedure can call the command utility subroutine (cu_) to obtain its arguments and to get other information about the command environment. A command procedure can call the command error subroutine (com_err_) to report errors to the user. The example below shows the portion of a command procedure that obtains its argument count and scans the arguments looking for an "-input" control argument.

```
     sample_command:  procedure options (variable)

declare arg_count fixed binary (17);
declare arg_len fixed binary (21);
declare arg_ptr pointer;
declare arg character (arg_len) based (arg_ptr);
declare code fixed binary (35);
declare argx fixed binary (17);
declare cu_$arg_count entry (fixed binary (17), fixed binary (35));
declare cu_$arg_ptr entry (fixed binary (17), pointer,
      fixed binary (21), fixed binary (35));
declare com_err_ entry options (variable);
     .
     .
call cu_$arg_count (arg_count, code);
if code ^= 0 then do; /* not invoked as command */
     call com_err_ (code, "sample_command");
     return;
end;
     .
     .
do argx = 1 to arg_count;
     call cu_$arg_ptr (argx, arg_ptr, arg_len, (0));
     if arg = "-input" then do;
     .
     .
end;
     .
     .
end sample_command;
```

Detailed information about the command utility and command error subroutines is provided in the Subroutines manual.

Writing an Active Function

Active functions are special command procedures that return a value to the command processor. The command processor substitutes this value into the command line in place of the active string that caused the active function to be called.

Active function procedures differ from other procedures in the following ways:

1. An active function procedure can receive only nonvarying character-string arguments.

2. An active function procedure can only receive input arguments. An error may occur if the procedure changes any of its input arguments.

3. An active function procedure must be prepared to handle a variable number of input arguments.

4. An active function procedure returns a value in a varying character string provided by the command processor. The active function may assign any character string value (including a null character string) to this return string. When the active function procedure returns, the command processor substitutes the value of the return string in place of the active string which caused the active function procedure to be called.

An active function procedure can call the command utility subroutine (cu_) to obtain its input arguments and return string from the command processor. It can call the active function error subroutine (active_fnc_err_) to report errors to the user. The example below shows the portion of an active function procedure that obtains its return string and a count of its input arguments. The active function reports a command error if it was not called as an active function. It expects no input arguments and therefore reports an error if any were given in the active string.

```
sample_active_function: procedure;
declare arg_count fixed binary(17);
declare return_stringl fixed binary(21);
declare return_stringp pointer;
declare return_string character(return_stringl) varying
     based (return_stringp);
declare code fixed binary(35);
declare error_table_$too_many_args fixed binary(35)
     external static;
declare cu_$af_return_arg entry (fixed binary(17), pointer,
     fixed binary(21), fixed binary(35));
declare (active_fnc_err_, com_err_) entry options(variable);
     .
     .
     .
call cu_$af_return_arg (arg_count, return_stringp,
     return_stringl, code);
if code ^= 0 then do:    /* error if called as a command,
                            not as an active function.  */
     call com_err_ (code, "sample_active_function");
     return;
     end;
if arg_count ^= 0 then do; /* error if any args given. */
     call active_fnc_err_ (error_table_$too_many_args,
      "sample_active_function", "No arguments expected.");
     return;
     end;
return_string = "";  /* initialize return string.  */
     .
     .
     .
end sample_active_function;
```

Detailed information about how the command utility and active function error subroutines can be used from an active function procedure is provided in the Subroutines manual.

The same procedure can be programmed to operate both as an active function and as a command procedure. Typically when such procedures are called as a command, they print on the user's terminal the value of the string they would return as an active function. These command/active function procedures are coded as active functions and should call cu_$af_return_arg instead of cu_$af_arg_count. If cu_$af_return_arg returns the error code error_table_$not_act_fnc, they operate as commands. If the code returned is zero, they use the returned pointer and length to base the return value. Any other nonzero error code should be fatal. Note that cu_$af_return_arg always returns a correct argument count even if the active function was invoked as a command, so the user can go on to use cu_$arg_ptr with no further checking.

ADDRESS SPACE MANAGEMENT

When a user logs in, he or she is assigned a newly created process. Associated with the process is a collection of segments that can be referenced directly by system hardware. This collection of segments, called the address space, expands and contracts during process execution, depending on which segments are used by the running programs.

Address space management consists of constructing and maintaining a correspondence between segments and segment numbers, segment numbers being the means by which the system hardware references segments. Segment numbers are assigned on a per-process basis (i.e., for the life of the process), by supplying the pathname of the segment to the supervisor. This assignment is referred to as "making a segment known." Segments are made known automatically by the dynamic linker when a program makes an external reference; making a segment known can also be accomplished by explicit calls to address management subroutines. In addition, when a segment is made known, a correspondence can be established between the segment and one or more reference names (used by the dynamic linker to resolve external references); this is referred to as "initiating a reference name." When dynamic linking is the means used to make a segment known, the initiation of at least one reference name is performed automatically. (For more information on reference names, see "Reference Names" in Section 3 and "Making a Segment Known" below.) A general overview of dynamic linking is given below.

Dynamic Linking

The primary responsibility of the dynamic linker is to transform a symbolic reference to a procedure or data into an actual address in some procedure or data segment. In general, this transformation involves the searching of selected directories in the Multics storage system and the use of other system resources to make the appropriate segment known. The search for a referenced segment is undertaken after program execution has begun and is generally required only the first time a program references the address.

The dynamic linker is activated by traps originally set by the translator in the linkage section of the object segment. These traps are used by instructions making external references. When such an instruction is encountered during execution, a fault (trap) occurs and the dynamic linker is invoked.

The dynamic linker uses information contained in the object segment's definition and linkage sections to find the symbolic reference name. (For a detailed description of these sections, see Appendix G.) Using the search rules currently in effect, the dynamic linker determines the pathname of the segment being referenced and makes that segment known. The linkage trap is modified so that the fault does not occur on subsequent references; this is referred to as snapping the link.

Search Rules

In order to resolve external references, the dynamic linker uses a prescribed search list specifying a subset of the directory hierarchy. The search for a segment proceeds as follows. If the reference name is found in the list of initiated segments (item 1 below), that segment is used. Otherwise, directories are searched in the order in which they appear in the search rules until the name is found. The standard search rules are given below. These can be modified using the add_search_rules, delete_search_rules, and set_search_rules commands (described in the Commands manual). The installation may also modify the default search rules for all users by using the set_system_search_rules command, described in the *Multics Administration, Maintenance, and Operations Commands* manual, Order No. GB64. The get_system_search_rules command, described in the Commands manual, prints the current system default search rules.

1. initiated segments

   Reference names for segments that have already been made known to a specific process are maintained by the system. A reference name is associated with a segment in one of three ways:

   a. use in a dynamically linked external program reference

   b. a call to hcs_$initiate, or hcs_$make_seg with a nonnull character string supplied as the ref_name argument (these hcs_ entry points are described in the Subroutines manual)

   c. a call to hcs_$make_ptr (described in the Subroutines manual)

2. referencing directory

   The referencing directory contains the procedure segment whose call or reference initiated the search.

3. working directory

   The working directory is the one associated with the user at the time of the search. This may be any directory established as the working directory by either the change_wdir command or the change_wdir_ subroutine (described in the Commands and Subroutines manuals respectively). (The initial working directory is the home directory.)

4. system libraries

The system libraries are searched in the following order:

>system_library_standard
This library contains standard system service modules, i.e., most system commands and subroutines.

>system_library_unbundled (if present)
This library contains unbundled software.

>system_library_1
This library contains a small set of subroutines that are reloaded each time the system is reinitialized.

>system_library_tools
This library contains software primarily of interest to system programmers.

With the search rules given above, when a program in the user's working directory has the same name as a system program, the user program will be invoked (since it is found first). Unless this is intended, the user should avoid using the names of system commands for his programs, or should change either his working directory or the search rules in effect. (An exception to this occurs if the reference is by a program in the same directory as the system program being searched for; see item 2, above.) If an external reference to a procedure is not resolved by following the search rules, an error message is printed. The user can recover from the error in a number of ways (for example, by initiating the procedure directly or by adding a link to the procedure into one of the directories included in the search rules).

Binding

Binding is an alternative to dynamic linking that should be used when a set of object segments is intended to be executed together repeatedly. Using the bind command, a user can consolidate these segments into a single bound object segment. Binding can provide a substantial savings in processing time and page fault overhead.

Binding proceeds as follows. The object code portions of the segments to be bound are concatenated and relocated as necessary. Intersegment references are resolved with direct text-to-text or text-to-internal-static references within the bound segment components. A new set of definitions and linkage information is created to reflect the interface between a bound segment and external references. (For more details on binding, see the bind command in the Commands manual; for the structure of a bound segment, see Appendix G of this manual.)

Making a Segment Known

A segment is known to a process when it has been uniquely associated with a segment number in that process. This association is maintained for the life of the process unless a user explicitly makes the segment unknown.

Once a segment is known by a given segment number, all program references using that number are interpreted by the system hardware and associated software as references to that segment. A segment can be made known through dynamic linking or by explicit calls.

A segment can be made known without a reference name, through the initiate_file_ subroutine (the terminate_file_ subroutine makes it unknown). These subroutines are described in the Subroutines manual. When a segment is made known, a reference name can be associated with it. Such a name is said to be initiated for the segment (see the hcs_$initiate_ entry point in the Subroutines manual). The association between a reference name and a segment lasts as long as the segment is known unless explicitly discontinued by the user. The ending of this association is referred to as terminating the reference name. A segment may be initiated by more than one reference name, but no two segments can have the same reference name.

Reference names that have been initiated are the first items examined by the dynamic linker (see "Search Rules" above) when attempting to find a referenced procedure or data segment. If the name is not initiated, the dynamic linker makes the segment known and initiates that name for the segment when it has successfully completed its search.

The user can remove reference names by using the term_ subroutine (described in the Subroutines manual). If only one reference name appears for a segment and it is terminated, the segment is also made unknown. The user may also explicitly make a segment unknown and terminate all its reference names (see term_$seg_ptr in the Subroutines manual).

At command level, the initiate and terminate commands may be used to initiate and terminate reference names. (See the Commands manual for a discussion of these commands.)

Address Space Management Subroutines

The subroutines listed below provide a direct interface between user-written programs and some of the system mechanisms discussed previously. The selection of the appropriate routine is based on the form in which the segment of interest is currently expressed. For example, if an interactive program accepts the pathname of a segment as an argument, that segment can be made known using initiate_file_.

A brief description of these interface subroutines is given below. For a complete description, see the Subroutines manual.

hcs_$fs_get_path_name
: given a pointer to a segment, returns its pathname

hcs_$fs_get_ref_name
: given a pointer to a segment, returns associated reference names

hcs_$fs_get_seg_ptr
: given a reference name, returns a pointer to the associated segment

initiate_file_
: given a pathname, causes the segment to be made known.

hcs_$initiate
: given the pathname of a known segment and a reference name, initiates the reference name.

hcs_$make_entry
: given a reference name and the name of an entry point, returns the value of the specified entry point.

hcs_$make_ptr
: given a reference name and the name of an entry point, returns a pointer to the specified entry point. If the reference name is not yet initiated, search rules are used to find a segment with the same name, the segment is made known and the reference name initiated.

term_
: given a pathname, terminates all reference names of a segment and makes it unknown.

term_$single_reference
: terminates one reference name from a segment. If it is the only reference name for that segment, the segment is made unknown.

terminate_file_
: given a pointer to a segment, makes the segment unknown if there are no reference names associated with the segment.

term_$seg_ptr
: given a pointer to a segment, terminates all reference names and makes the segment unknown.

MULTICS STACK SEGMENTS

The Multics stack segment is a central component of the normal execution environment. It is essentially a pushdown list where active procedures maintain private regions, called stack frames, in which their temporary variables reside. A stack frame is created for a procedure when it is called; the procedure is subsequently referred to as the owner of the stack frame. Stack frames also contain information used in interprocedure communication, such as argument lists and procedure return points. The base of the stack segment, the stack header, contains pointers to various types of information about the process. Elements of the stack are described briefly below and in detail in Appendix H.

Stack Header

The stack header contains pointers to code sequences (used to perform the standard procedure call and return and stack push and pop functions) and to operator segments (containing brief code sequences referenced by programs compiled by system translators). Another set of pointers is maintained to keep track of the stack frames created and released during the process. Two pointers in the stack header are used to implement external reference resolutions on an interprocedure and intersegment basis. These point to the linkage offset table (LOT) and the internal static offset table (ISOT) for the current ring. The LOT points to the dynamic linkage sections allocated in the ring and the ISOT to the dynamic internal static sections allocated in the ring.

Stack Frames

The stack frame is used to store the current state of the calling procedure and the information used to restore that state when a return from the call is made. The stack frame also contains data associated with the procedure to be executed. The stack frame header contains pointers to information required to activate the called procedure, such as a pointer to the argument list and to the linkage region of the calling procedure. Since a new stack frame is generally created at each call, procedures that have variables in the stack frame are potentially recursive.

Combined Linkage Region

A combined linkage region can consist of one or more segments that contain a sequence of contiguous linkage sections (pointed to by the LOT), internal static sections (pointed to by the ISOT), or general storage regions acquired through system routines for all object segments active in the ring. Additional segments are created as necessary to contain this information.

CLOCK SERVICES

Two types of clocks are available on Multics: a real-time clock for the entire system and a process execution timer for each process. The real-time clock, a hardware calendar clock accessible through a special register on a system controller, runs whenever the system is in operation; it contains a double-word integer register that is incremented once per microsecond and represents the number of microseconds elapsed since January 1, 1901, 0000 hours Greenwich mean time. A simulated interrupt

mechanism is associated with the calender clock so that a specified process can receive an interprocess wakeup at any given time.

A process execution timer is maintained as part of the state of each process. It counts the microseconds used by a process. This timer measures virtual CPU time (in microseconds) spent by the process. In addition, it can be used for setting timed wakeups.

An interrupt mechanism associated with the virtual timer allows a process to receive an interprocess wakeup after a given amount of CPU time has been used. The timer is compared to the specified value at regular intervals; when the value is exceeded, an interprocess wakeup is generated for the running process.

The clocks are available for use by programmers. Some ways in which system commands use them are given below:

- Resource monitoring and accounting.
- Labeling data (e.g., storage system entries) with dates and times of interest.
- Computing the date and time for output.
- Generating a unique bit string.
- Waking up a specified process at a specified time, perhaps causing a specified procedure to be called.
- Interrupting a process after a specified amount of CPU time has elapsed.

Access to System Clocks

Commands and subroutines that permit the user to inspect the real-time clock and the process execution timer are summarized below. For a detailed description of each, see the Subroutines manual. The Multics PL/I built-in function clock() reads the real-time clock and return its current value as a fixed bin(71) quantity. This clock time can be converted to a more readable form using either date_time_, which returns a single character string, or decode_clock_value_, which returns the various components of the time (month, year, etc.) as distinct variables. The convert_date_to_binary_ subroutine accepts a character string like that produced by date_time_ and returns a fixed bin(71) equivalent. The set_time_zone command enables the user to set the default time zone for his or her process.

The value of the process execution timer is returned by both the cpu_time_and_paging_ and the Multics PL/I built-in function vclock(). The resource_usage command (described in the Commands manual) prints a report of the resources used by the user from the beginning of the current billing period to the time of creation of the user's current process.

The status command and the hcs_$status_ subroutine both provide dates and times associated with storage system entries, such as the date and time the entry was last modified and the date and time last used. The hcs_$status_ subroutine returns the time in file system format; this 36-bit time can be converted by the contruct:

```
clock_value = cv_fstime_ (fstime);
```

The unique_bits_ subroutine returns a bit string, generated partly from the current real-time clock reading, that is guaranteed to be unique among all bit strings so generated. The unique_chars_ subroutine converts such a value into a character string that is also guaranteed to be unique among all character strings so generated.

Facilities for Timed Wakeups

The interprocess communication facility (see the ipc_ subroutine in the Subroutines manual) allows a user to set up channels for sending interrupts (wakeups) to a specified process. The interrupt can cause that process to return from the blocked state to whatever it was previously doing, or it can cause some other procedure to be called in that process. One possible use of this facility is to wake up a process as the result of some clock activity. The timer_manager_ subroutine (described in the Subroutines manual) provides the necessary interface. With this subroutine, the user can specify an event channel for his own or another process, whether the process should merely be wakened or a specified procedure should be called, and the nature of the clock activity that should trigger the wakeup (i.e., virtual CPU or calendar clock time). In specifying the time, the user can further specify absolute or relative time and can use seconds or microseconds.

WRITING A PROCESS OVERSEER

Almost every feature of the standard Multics system interface can be replaced by providing a specially tailored process overseer procedure in place of the standard version. The standard Multics process overseer procedure, process_overseer_, is the initial procedure assigned to a user unless the project administrator specifies otherwise by an initproc or Initpr oc statement in the project master file (PMF). (See the *Multics Administartors' Project,* Order No. AK51.) If a user has the v_process_overseer attribute, she may specify a different initial procedure when she logs in by using the -process_overseer (-po) control argument as in the following example:

```
login Smith -po >udd>AEC>special_overseer_
```

If Smith does not have the v_process_overseer attribute, the system refuses the login.

If the user has the v_process_overseer attribute, she may leave a program named "process_overseer_" in her homedir. Note that if the PMF specifies a reference-name other than "process_overseer_", the user must put whatever it specifies in her homedir. If the PMF provides an absolute pathname for the initial procedure, the user can not replace it in this manner.

Process Initialization

A process is created for a user when she logs in, or in response to either a new_proc command (described in the Commands manual) or process termination signal. What follows is a brief description of the birth of a process.

Unless otherwise noted, all of the modules described are in PL/I. It is helpful to follow along this discussion with a listing of the modules; the comments often provide useful amplification. To do so, use the library_fetch command. For example:

```
lf initialize_process_.pl1
```

Several items of information must be passed to all processes by the system control process. The system places this information in a special per-process segment, called the process initialization table (PIT), that resides in the process directory. The user process may read the contents of the PIT, but may not modify it because its write bracket is zero. The user_info_ subroutine (described in the Subroutines manual) is used to extract information from the PIT.

A process begins, for all intents and purposes, with a call to the ring zero routine init_proc. This description will only mention those actions of init_proc which are of significance to visible features of the user environment.

The first action of init_proc is to initialize the known segment table (KST) by calling initialize_kst. Then init_proc initializes the PIT, and checks for the v_process_overseer attribute. If v_process_overseer is on, init_proc sets the working directory to the user's home directory. Until this point the user has no working directory, so that users without v_process_overseer do not get their home directory into the search rules until later on in their process. This prevents users without v_process_overseer from replacing their initial procedure, signaller, or unwinder.

Now init_proc calls makestack to create the stack in the user's initial ring. First, makestack creates a segment named stack_N in the process directory, where N is the number of the user's initial ring. It fills in the null pointer, begin pointer, and end pointer of the stack and calls the linker (via link_man$get_initial_linkage), to get the initial linkage for the ring.

The internal procedure initialize_rnt is then called by makestack in order to make a reference name table (RNT) for the ring in question. initialize_rnt calls define_area_ to get an area for the RNT, and puts a pointer to the RNT into the appropriate place in the stack header. Then initialize_rnt initializes the search rules to the default rules and returns.

At this point makestack adds the name of the stack it is creating to the RNT and calls the linker to snap links to signal_, unwinder_, the alm operators, and pl1_operators_. Thus users with v_process_overseer, whose working directories were set by init_proc before makestack was called, pick up any versions of these programs that are resident in their home directories. It then sets up the static condition handlers for no_write_permission, not_in_write_bracket, isot_fault, and lot_fault, fills in the thread pointers for the first stack frame and returns.

Now, init_proc is ready to find the initial procedure. For the purposes of this discussion, the initial procedure is the first procedure called in the user's initial ring. The term "process overseer" will refer to the program specified by the initproc keyword of the PMF or the argument to the -process_overseer control argument of the login access request. If the string ",direct" is appended to the pathname specified by either the initproc keyword or the -process_overseer control argument, then the specified pathname is both the process overseer and the initial procedure and init_proc parses the pathname and initiates it explicitly. This is because link_snap$make_ptr (the ring 0 entry that snaps links) will not take absolute or relative pathnames. Therefore init_proc parses the supplied pathname as either an absolute pathname or a relative pathname relative to the user's home directory. Note that this is independent of the state of v_process_overseer -- if the project administrator specified a ,direct overseer with a relative pathname, it will reference off of the home directory. This primarily provides a typing convenience to users with v_process_overseer specifying a ,direct overseer at login. If the name does not end with ,direct, the standard initial procedure, initialize_process_, is used.

At this point init_proc either has a pointer and a reference name for a ,direct overseer, or it has a reference name to the standard initial procedure initialize_process_.

Finally, init_proc calls call_outer_ring_ to call out to the user's initial ring. Note that a user without v_process_overseer is still lacking a working directory. It is the responsibility of any user-supplied ,direct initial procedure to set the working directory.

The user's process now begins execution in the initial ring in the program initialize_process_.

The initialize_process_ procedure first initiates the PIT. If the user lacks v_process_overseer it finds the appropriate process overseer. Then it sets the working directory, and finds the process overseer if it was not previously found. It sets up static condition handlers for cput, alrm, trm_, wkp_ and sus_.

Before calling the process overseer, initialize_process_ attaches the I/O switch named user_i/o (through an I/O system module named in the PIT) to the target (also specified in the PIT). It then attaches the I/O switches named user_output, user_input, and error_output as synonyms of user_i/o by calling iox_$init_standard_iocbs. The I/O module used for an interactive process is tty_, the Multics terminal device I/O module. (This module is described in the Subroutines manual). For absentee processes it is abs_io_, and for daemons it is mr_.

Absentee processes do not use any of the login arguments or attributes of the process which submitted the absentee request. All absentee process attributes come only from the absentee request, the system administrator table (SAT), and the project-definition table (PDT).

The initialize_process_ procedure then calls the process overseer specified in the PIT. This is either the procedure specified in the "initproc" keyword of the PMF, or the -po argument to login. It is called with the following arguments:

```
declare process_overseer_ entry (ptr, bit (1) aligned, char (*)
     varying);

call process_overseer_ (pit_ptr, call_listen_, initial_cl);
```

where:

1. pit_ptr
   is a pointer to the PIT. (Input) It should be ignored.

2. call_listen_
   if set to "1"b, initialize_process_ will call listen_ with the value of initial_cl as the first command line, thus starting the command environment. (Output) If it is set to "0"b , the process will be terminated, on the assumption that the process overseer already ran the entire process.

3. initial_cl
   is the first command line to be executed, normally an exec_com of the start_up ec. (Output) It may be up to 256 characters long.

Process Overseer Functions

The system process overseers terminate processing by setting the call_listen flag in their calling sequence, setting the initial_cl argument to the initial command line, and returning to initialize_process_.

A user-supplied process overseer procedure may perform many other actions besides those executed by the system version. For example, initialization of special per-project accounting procedures may be accomplished at this point, or requests issued for an additional password or any other administrative information required by a project.

The initial command line used by the system process overseer is:

```
exec_com start_up_path>start_up.ec start_type proc_type
```

where:

1. start_up_path
   is the location of the user's start_up.ec. The system process overseers search for the start_up.ec in the following directories, in this order:

   ```
   >udd>Project>person, >udd>Project, and >system_control_1.
   ```

2. start_type
   is either login or new_proc, depending on which of these was invoked to create the process.

3. proc_type
   is either interactive, absentee, or daemon.

These arguments can be used by the start_up.ec segment as described in connection with the exec_com command in the Commands manual.

The command line given above assumes that the no_start_up flag is off and that the segment named start_up.ec can be found. The no_start_up flag is off unless the project administrator has given the user the no_start_up attribute and the user has included the proper control argument (-no_start_up or -ns) in his login line.

If the process overseer returns to initialize_process_ with the call_listen flag set, initialize_process_ establishes an any_other handler of default_error_handler_$wall by executing the statement:

```
on any_other call wall_entry_variable;
```

An entry variable is used because initialize_process_ calls hcs_$make_entry with a null referencing pointer, so that users with v_process_overseer can put private versions of default_error_handler_ in their homedirs.

The default_error_handler_$wall procedure is invoked on all signals not intercepted by any subsequently established condition handler. In general, the default_error_handler_$wall procedure either performs some default action (such as inserting a pagemark into the stream when an endpage condition is signalled) and restarts execution, or else it prints a standard error message and calls the current listener.

If the process overseer does not use the call_listen_ flag, it must establish its own any_other handler, and call the listener if cleared.

Some Notes on Writing a Process Overseer

The best source of information on the writing of process overseers is the source of the standard one: process_overseer_.pl1. There are, however, several important considerations not obvious from the source.

The first is that process_overseer_ makes use of the pointer to the PIT that it gets as an argument. This means that if the PIT format changes, at best process_overseer_ must be recompiled. At worst, it may have to be recoded. If a user process overseer uses the PIT instead of calling user_info_, then it will likely stop working if the format of the PIT changes. For this reason, we strongly recommend that user-written process overseers do not directly reference the PIT. They should call user_info_, instead.

Both of the installed process overseers look for start_up exec_coms. The process_overseer_ and project_start_up_ procedures try to find start_up.ec in the home directory, the project directory, and >sc1 before giving up. Privately written process overseers should do so as well, unless they are putting the user in an environment for which this is obviously inappropriate.

Direct Process Overseers

The ,direct overseers are called as the first procedure in the user ring. In addition to setting up all I/O attachments for user_i/o, and static condition handlers for alrm, cput, trm_, wkp_ and sus_, ,direct overseers are responsible for setting the working directory for users without v_process_overseer. This is done to make protection somewhat easier, as the direct overseer can find anything it is interested in before setting the working directory.

Handling of Quit Signals

A quit signal is indicated by pressing the appropriate key, such as ATTN or BRK, on the terminal in use. When a terminal is first attached for interactive processing, quit signals from the terminal are disabled. A user quit signal issued at this time causes the flushing of terminal output buffers, but the quit condition is not raised in the user ring. The recognition of quit signals is enabled when the following call is made:

```
call iox_$control (iox_$user_io, "quit_enable", null(), status);
```

If a project administrator wishes to replace the standard user environment with his own programs, he must find an appropriate place for the quit_enable order, after the mechanism for handling quit signals has been established.

CREATING AN EXTENDED ENTRY

An extended entry is a storage system entity which is created and manipulated by a particular subsystem and for which the operations performed by the standard file system commands and subroutines are either incorrect or impossible. For example, the mailbox is created and maintained by the message segment facility. All mailboxes must be named with the suffix ".mbx" and their accessibility is defined by extended modes instead of the more familiar "rew" or "sma" modes. In addition, proper access to these entries is enforced by the fact that they are ring one resident; they are inaccessible from the user ring unless message_segment software (ring one resident) is invoked through a gate.

Extended entry software allows the standard Multics commands and subroutines to operate upon extended entries without compromising the integrity of the owning subsystem. This applies not only to system-supplied extended entries such as mailboxes and message segments, but also to user-written subsystems.

A number of commands and subroutines have been modified to correctly handle extended entries. These include add_name, copy, copy_acl, copy_dir, copy_names, delete, delete_acl, delete_name, list_acl, move, move_dir, rename, set_acl, set_max_length, set_ring_brackets, status, switch_on, switch_off, copy_, copy_dir_, copy_acl_, copy_names_, delete_, dl_handler_, nd_handler_.

There are two new commands for printing information about extended entries. The list_entry_types (lset) command will print a list of all the entry types that can be found in the search rules. It provides only the name of the entry type and the suffix it uses. For detailed information about an entry type, the describe_entry_type (dset) command can be used. It prints out the name of the type, various attributes, and the pathname of an info segment containing more information about the entry type. The status commands accepts a "-type" control argument which prints the type of a storage system entry, be it extended (as for mailboxes) or standard (segment, directory, etc.). All three of these commands also work as active functions.

The extended entry facility is implemented by the fs_util_ subroutine. The entry points in fs_util_ are used to change and retrieve information about any file system entry. The fs_util_ subroutine determines whether or not the entry is extended by examining its suffix. Assuming that the suffix is XXX, fs_util_ attempts to locate and then invoke a subroutine named "suffix_XXX__$validate". If both of these actions succeed, the entry in question is considered to be extended and subsequently referenced through suffix_XXX_; otherwise it is considered standard and fs_util_ decides how to reference it based upon its standard entry type (link, segment, directory, multisegment file, or data management file).

In order to enable an application program to operate upon extended entries, the hcs_ calls it makes to perform file system operations should be replaced by the corresponding fs_util_ calls. Calls to any one of the standard system subroutines listed above, which already support extended entries, need not be changed.

To implement an extended entry type, you must choose a suffix and implement a suffix support routine for that entry type. Each suffix support routine must support the suffix_info and validate entrypoints, as well as a few optional ones. These routines will be called by fs_util_. If any of the optional entry points are not provided, fs_util_ returns an error code of error_table_$unsupported_operation to its caller and the particular operation fails.

There are two restrictions in implementing an extended entry type. First, an extended entry type may use only standard entry types (segment, directory, multisegment_file, and dm_file), as the underlying storage type. Second, if the acl modes for the extended entry type do not map directly onto the standard modes, then they must be stored in the extended modes field of the underlying type. For example: mailboxes use the extended modes field of segments to store their modes of "adrosw".

The following is a list of supported entrypoints for suffix support routines:

```
dcl suffix_XXX_$add_acl_entries entry (char(*), char(*), ptr,
      fixed bin(35))

call suffix_XXX_$add_acl_entries (dir_name, entry_name, acl_ptr,
      code)
```

NOTE: acl_ptr points to general_acl in acl_structures.incl.pl1.

```
dcl suffix_XXX_$chname_file entry (char(*), char(*), char(*),
      char(*), fixed bin(35))

call suffix_XXX_$chname_file (dir_name, entry_name, old_name,
      new_name, code)

dcl suffix_XXX_$copy entry (ptr, fixed bin(35))

call suffix_XXX_$copy (copy_options_ptr, code)
```

NOTE: copy_options_ptr points to the structure copy_options defined in copy_options.incl.pl1. This routine only copies the contents of an entry, directories excluded. copy_ may be used copy other attributes as well as the contents.

```
dcl suffix_XXX_$delentry_file entry (char(*), char(*), fixed
      bin(35))

call suffix_XXX_$delentry_file (dir_name, entry_name, code)

dcl suffix_XXX_$delete_acl_entries entry (char(*), char(*), ptr,
      fixed bin(35))

call suffix_XXX_$delete_acl_entries (dir_name, entry_name,
      acl_ptr, code)
```

NOTE: acl_ptr points to general_delete_acl in acl_structures.incl.pl1.

```
dcl suffix_XXX_$get_bit_count entry (char(*), char(*), fixed
      bin(24), fixed bin(35))

call suffix_XXX_$get_bit_count (dir_name, entry_name, bit_count,
      code)

dcl suffix_XXX_$get_max_length entry (char(*), char(*), fixed
      bin(19), fixed bin(35))

call suffix_XXX_$get_max_length (dir_name, entry_name, max_length,
      code)

dcl suffix_XXX_$get_ring_brackets entry (char(*), char(*), (*)
      fixed bin(3), fixed bin(35))

call suffix_XXX_$get_ring_brackets (dir_name, entry_name,
      ring_brackets, code)
```

NOTE: ring_brackets may be an array of 1, 2 or 3 elements.

```
dcl suffix_XXX_$get_switch entry (char(*), char(*), char(*),
      bit(1) aligned, fixed bin(35))

call suffix_XXX_$get_switch (dir_name, entry_name, switch_name,
      switch_value, code)

dcl suffix_XXX_$get_user_access_modes entry (char(*), char(*),
      char(*), fixed bin, bit(36) aligned, bit(36) aligned, fixed
      bin(35))

call suffix_XXX_$get_user_access_modes (dir_name, entry_name,
      user_name, ring, modes, extendeded_modes, code)
```

NOTE: ring is the validation level to be used in computing effective access. It may be a value between 0 and 7 inclusive, or -1. -1 may be used to specify the default value of the validation level of the calling process.)

```
dcl suffix_XXX_$list_acl entry (char(*), char(*), char(*), ptr,
      ptr, fixed bin(35))

call suffix_XXX_$list_acl (dir_name, entry_name, version,
      area_ptr, acl_ptr, code)
```

NOTE: area_ptr points to the start of the allocated area; if null, system_free_area will be used. acl_ptr points to the general_acl structure in acl_structures.incl.pl. If acl_ptr is nonnull or input, general_acl.version and general_acl.access_name (*) are set indicating which ACL entries to list. If acl_ptr is null, the entire acl for the branch must be listed.

```
dcl suffix_XXX_$list_switches entry (char(*), ptr, ptr, fixed
      bin(35))

call suffix_XXX_$list_switches (version, area_ptr,
      switch_list_ptr, code)
```

NOTE: switch_list_ptr points to switch_list structure; version must be SWITCH_LIST_VERSION_1. Both are declared in suffix_info.incl.pl1.

```
dcl suffix_XXX_$replace_acl entry (char(*), char(*), ptr, bit(1),
      fixed bin(35))

call suffix_XXX_$replace_acl (dir_name, entry_name, acl_ptr,
      no_sysdaemon_sw, code)
```

NOTE: acl_ptr points to general_acl structure in acl_structures.incl.pl1. no_sysdaemon_sw specifies whether or not access for *.SysDaemon should be added to the entry's acl.

```
dcl suffix_XXX_$set_bit_count entry (char(*), char(*), fixed
      bin(24), fixed bin(35))

call suffix_XXX_$set_bit_count (dir_name, entry_name, bit_count,
      code)

dcl suffix_XXX_$set_max_length entry (char(*), char(*), fixed
      bin(19), fixed bin(35))

call suffix_XXX_$set_max_length (dir_name, entry_name, max_length,
      code)

dcl suffix_XXX_$set_ring_brackets entry (char(*), char(*), (*)
      fixed bin(3), fixed bin(35))

call suffix_XXX_$set_ring_brackets (dir_name, entry_name,
      ring_brackets, code)

dcl suffix_XXX_$set_switch entry (char(*), char(*), char(*),
      bit(1) aligned, fixed bin(35))

call suffix_XXX_$set_switch (dir_name, entry_name, switch_name,
      switch_value, code)

dcl suffix_XXX_$suffix_info entry (char(*), char(*), ptr, fixed
      bin(35))

call suffix_XXX_$suffix_info (dir_name, entry_name,
      suffix_info_ptr, code)
```

NOTE: Fill in the appropriate values in suffix_info structure in suffix_info.incl.pl1.

```
dcl suffix_XXX_$validate entry (char (*), char(*), fixed bin (35))

call suffix_XXX_$validate (dir_name, entry_name, code)
```

NOTE: A 0 error code must be returned if the entry is a valid XXX; otherwise, error_table_$not_seg_type should be returned.

If the get_switch, set_switch and list_switches entrypoints are present, the switch_on and switch_off commands may be used to change the value of those switches, and the status command may be used to report them.

All entry points in a suffix_XXX_, which manipulate entries, with the exception of validate and suffix_info, are responsible for validating that the entry is actually of the correct type an not merely masquerading as one be having the appropriate suffix. If the entry is not of the correct type, the error code error_table_$not_seg_type should be returned.

Two fs_util_ entry points may be of help when writing a suffix support routine or an application that may use extended entry types.

```
dcl fs_util_$get_type (char(*), char(*), char(*), fixed bin(35))

call fs_util_$get_type (dir_name, entry_name, type, code);
```

where TYPE (an output argument) may be one of the standard or extended entry types. The standard entry types are as constants in suffix_info.incl.pl1.

```
dcl fs_util_$make_entry_for_type entry (char(*), char(*), entry,
      fixed bin(35))

call fs_util_$make_entry_for_type (TYPE, OPERATION, entry_to_call,
      code);
```

where TYPE (an input argument) is as above, and OPERATION (also an input argument) is the name of the operation for which an entry is requested. The entry_to_call may the be used to operate on any entry. It will have a calling sequence identical to the corresponding fs_util_ entrypoint. A list of designated constants for the names of operations is available in file_system_operations.incl.pl1.

If you wish to implement an extended type named "chess_game" that uses a segment as its underlying storage type, and wish to use all the switches that segments do, the entrypoint to set switches would be implemented thusly:

```
set_switches: entry (dir_name, entry_name, switch_name, switch_value,
                                                          code);
call VALIDATE_CHESS_GAME ();  /* an internal procedure */

call fs_util_$make_entry_for type (FS_OBJECT_TYPE_SEGMENT,
     FS_SET_SWITCH, set_switch_entry, code);
call set_switch_entry (dir_name, entry_name, switch_name,
switch_value, code);

return;
```

INTERACTIVE SUBSYSTEM PROGRAMMING ENVIRONMENT

The Multics Subsystem Utilities provide a general-purpose interface for implementing interactive subsystems such as the Extended Mail Facility and Forum. The ssu_ subroutine is the vehicle for implementing interactive subsystems (see the Subroutines manual); the Subsystem Utilities are referred to collectively as the utilities.

An interactive subsystem presents an interface analogous to Multics command level. After being invoked, the subsystem enters a request loop, where it prompts for and reads a request line, executes requests in the line, and optionally prints a ready message.

A subsystem normally consists of a command procedure and one or more request procedures. The command procedure is a Multics command or perhaps a process overseer that establishes the environment necessary for operation of the subsystem before entering the request loop. Individual requests, defined in request tables, are implemented as separate procedures invoked by the request processor. Request procedures process their arguments and report errors in a fashion similar to Multics commands.

To ensure a consistent interface to the subsystem user, the utilities perform several functions common to interactive subsystems, including maintenance of subsystem invocations and the request loop process. For example, most users are probably familiar with the mechanics of the Extended Mail Facility: how the prompt is issued; what to do for assistance (the various help facilities available); implications of the standard requests (list_requests, print, abbrev, help, etc.); how to exit from the subsystem (by issuing a quit request). In fact, the subsystem writer can design and implement an interactive subsystem that mirrors exactly a subsystem such as the Extended Mail Facility. The writer also has the option, however, of overriding many of the standard mechanisms, if the application warrants such action (see "Tailoring the Subsystem Environment" below).

The descriptions that follow document the procedures involved in implementing an interactive subsystem. The entry points cross-referenced in the text are all documented in the ssu_ subroutine description in the Subroutines manual.

Subsystem Invocations

A subsystem invocation usually corresponds to a single invocation of the subsystem's command procedure; e.g., read_mail. The objective of an invocation is to operate on a specific entity; e.g., an invocation of read_mail operates on a mailbox. Multiple active invocations can coexist; e.g., a subsequent invocation of read_mail can be invoked within read_mail (for this reason, subsystems should be written so that they use no static data). The command procedure parses the command line arguments, retrieves global options, and in general performs whatever initialization necessary to establish the subsystem invocation.

The invocation is created by a call to ssu_$create_invocation from within the command procedure. Subsequently, the listener is called, and it in turn calls the request processor when there are requests to be processed. Eventually, the user indicates a desire to exit from the subsystem (usually by issuing a quit request), and the listener returns to the command procedure, from which a call is made to ssu_$destroy_invocation, and the subsystem invocation ceases to exist.

The call to ssu_$create_invocation names the subsystem and provides a version identifier (e.g., to distinguish between the installed and experimental versions of a subsystem). It is also through ssu_$create_invocation that the cornerstones of communication within the subsystem are established: the info pointer (info_ptr) supplied as a parameter to the call, and the subsystem control info pointer (sci_ptr) returned by the call. These two pointers are discussed in greater detail below.

USE OF SCI_PTR AND INFO_PTR IN INTERACTIVE SUBSYSTEMS

When ssu_$create_invocation is called, it creates an internal data structure, pointed to by sci_ptr, and returns the pointer to the caller to be passed as a parameter in all other calls to the utilities. Such a data structure is created and maintained separately by the utilities for each invocation. Each data structure, which is transparent to the subsystem writer, contains everything needed by the utilities for the life of the invocation.

The info_ptr, on the other hand, must be supplied by the subsystem writer as a parameter to ssu_$create_invocation; the info_ptr is subsequently passed as a parameter in all calls to request procedures. The data structure it points to must be defined and maintained by the writer. It might contain, for example, pathnames, option switches, and pointers to global data bases needed during the invocation.

Although not a requirement, a recommended convention is to include the sci_ptr in the data structure pointed to by the info_ptr (the obverse is true; i.e., the data structure pointed to by sci_ptr contains the info_ptr). Then, only info_ptr need be passed between the procedures comprising the subsystem.

STAND-ALONE INVOCATIONS

A special form of subsystem invocation is available to the subsystem writer who may want to write a program that functions either as a Multics command or subsystem request. This is known as a stand-alone invocation and is created through a call to ssu_$standalone_invocation. Employing this method, the writer can greatly reduce the coding effort by eliminating parallel coding or numerous conditional constructs. Additional information appears below under "Subsystem Requests and Multics Commands."

Facilities are provided (ssu_$print_blast and ssu_$record_usage) for optionally tracking subsystem usage on a per-user basis. Usage statistics recorded include:

```
Person_id
Time of most recent use
Version used most recently
Number of times this subsystem used
Number of times this version used
Number of times this version announced
```

This information is recorded in a segment located by the linker search rules, using the ref_ptr (see ssu_$print_blast) to specify a referencing_dir. The segment, named `<subsystem_name>.ssusage`, must be explicitly created or no monitoring is performed. All users must be given rw access in order to record their usage statistics. If statistics cannot be recorded (no access, usage segment has not been created, segment is full), the version announcement mechanism is disabled (see below), and a nonzero code is returned. The return code is generally ignored since there is usually nothing to be done in this case. There are no consequences to being unable to record usage statistics (i.e., there are no penalties for unsuccessful calls to ssu_$print_blast and ssu_$record_usage).

As a subsystem undergoes change, it is assigned a version identifier to distinguish it from its predecessors (see the ssu_$create_invocation entry point). Since change often denotes functional improvements or different handling procedures, users need to be notified when a new version is installed and told of the nature of the changes. Notification is delivered in the form of a so-called blast message (see the ssu_$print_blast entry point), which appears the first N times the user invokes the new version of the subsystem (N is a threshold value specified through ssu_$print_blast). The blast message is disabled when statistics cannot be recorded, so that the user does not receive the message on every use of the subsystem.

Statistics can be tabulated without printing the blast message by calling ssu_$record_usage, which otherwise functions the same as ssu_$print_blast. Statistics are totaled for all users of a subsystem and can be viewed using the display_subsystem_usage command (see the Commands manual).

The Subsystem Environment

The subsystem environment consists of a cycle of events somewhat akin to a Multics process environment. The request loop is entered for the purpose of processing request lines just as the Multics command listener loop processes command lines. Request lines are formulated using a request language that emulates the command language. There are considerations in regard to the writing of requests, such as argument processing and error handling, that are also of concern in the writing of commands. Each of these areas as it pertains to subsystem writing is considered below in greater detail.

SUBSYSTEM REQUEST LOOP

The subsystem listener is called (ssu_$listen) from within the command procedure after the invocation has been established. The listener implements the request loop, which normally follows a specific pattern:

```
Print a prompt (the default)
Read a request line
Execute a request line
Print a ready message (default is no ready message)
```

The loop is eventually broken when a call is issued to ssu_$abort_subsystem (usually through a quit request), in which case the listener returns to the command procedure for any necessary housekeeping, before the invocation is destroyed and Multics command level is reinstated.

The default prompt string is a new line, followed by the subsystem name and optional invocation level number (e.g., if read_mail is called within read_mail, the second invocation appears as read_mail (2):), followed by a colon and two spaces. The default prompt is printed by a call to ioa_$nnl with the control string ^/<subsystem_name>^[(^d)^]:^2x (two additional arguments indicate whether the level number is greater than one, and what the level number is). The current prompt can be changed by calling ssu_$set_prompt; the ssu_$get_prompt entry point can be used to retrieve the current prompt string.

There also exists a set of prompt modes, which are essentially flags specifying whether: to prompt at all (default is on); to prompt after a blank line (default is off); to prompt after typeahead (default is on). Any or all of these modes can be changed by calling ssu_$set_prompt_mode and resetting the appropriate bits. A call to ssu_$get_prompt_mode returns the current bit settings.

Ready processing is off by default; it can be changed by a call to ssu_$set_ready_mode. A call to ssu_$get_ready_mode returns the current state of ready processing. If enabled, ready processing prints a ready message after executing each request. The default ready message is the same as that printed at command level. The ready message can be manipulated within the subsystem through the ready, ready_on, and ready_off requests (see Appendix J).

The listener itself reads the request line and calls the request processor (through ssu_$execute_line), which interprets the request line according to subsystem request language conventions, as described below. The listener also honors the escape to Multics command level convention (..). This feature is enabled by default (ssu_$cpescape), but can be disabled (ssu_$cpescape_disabled), if the application warrants such a restriction.

SUBSYSTEM REQUEST LANGUAGE

The subsystem request language is identical to Multics command language. The paragraphs below briefly review the language conventions. For a detailed description, see Section 3.

In the simplest case, a request line consists of a request name followed by optional arguments; the request name and arguments are separated from each other using whitespace (space, horizontal tab, etc.). Multiple requests may be invoked on a single request line by separating each request and its arguments from the others with a semicolon character. Arguments which contain whitespace or other characters recognized by the request line processor must be quoted by enclosing the argument within the quote character ("); if a quote is required as part of an argument, it must be doubled when placed within the enclosing quotes (e.g., "double""quotes" is the argument double"quotes). Request line iteration is specified through the use of parentheses. Active strings are specified through the use of brackets.

When used in request lines, active strings invoke active requests to obtain the values to be placed into the expanded request line. Active requests are the subsystem equivalent of active functions: they are subsystem requests which return a character string value. See "Defining Request Tables" below for descriptions of command requests and active requests.

The default request line processor provides facilities to tailor the request language in two distinct ways. First, the actual request language may be changed by enabling or disabling features such as iteration or by changing the characters used to invoke different features of the request language. Secondly, the user may request that abbreviations within request lines be expanded before actually executing the requests in the line. Both of these mechanisms are controlled by changing the current request processor options through use of the entry points ssu_$get_request_processor_options and ssu_$set_request_processor_options. Additionally, the abbreviation facility may be controlled by the user by using the standard subsystem request "abbrev" if it is defined within the subsystem.

If the standard subsystem request language is not suitable for a particular application even after changing the language definition as described below, the subsystem writer may implement a tailored request line processor to replace the standard processor. This mechanism is described below under "Tailoring the Subsystem Environment."

Modifying the Standard Request Processor

As mentioned above, the function of the standard request processor may be changed by modifying the request processor options in effect within the subsystem. The current request processor options are obtained by calling ssu_$get_request_processor_options. They are changed by calling ssu_$set_request_processor_options. The default request processor options for the subsystem invocation may be obtained by calling ssu_$get_default_rp_options. The current request processor options may be changed to their default settings by calling ssu_$reset_request_processor_options. For ease of reference in the remainder of this discussion, these entry points are referred to simply as get, set, get_default, and reset.

The default request processor options for a subsystem specify that the request language is identical to Multics command language and that abbreviations should not be expanded in request lines.

In order to change the request processor options, the subsystem should obtain a copy of the options currently in effect using the get entry point, modify this copy, and then call the set entry point.

For example, to change the request language to disable iteration, the following code fragment may be used:

```
dcl 1 local_rpo  aligned like rp_options;

     call ssu_$get_request_processor_options (sci_ptr,
          RP_OPTIONS_VERSION_1, addr (local_rpo), code);
     if code ^= 0 then call ssu_$abort_line (...);

     local_rpo.non_standard_language = "1"b;
     local_rpo.character_types (rank ("(")) = NORMAL_CHARACTER;
     local_rpo.character_types (rank (")")) = NORMAL_CHARACTER;
     call ssu_$set_request_processor_options (sci_ptr, addr
          (local_rpo), code);
     if code ^= 0 then call ssu_$abort_line (...);
```

The rp_options Structure

The request processor options are passed to and from the above mentioned entry points using the rp_options structure. For the get and get_default entry points, the contents of the structure are filled in to reflect the current or default request processor options. For the set entry point, the caller must fill in the contents of the structure to reflect the new options that are to be put into effect by the request processor. As the reset entry point always sets the options to their default state, it does not use this structure. In all cases where the structure is required, the caller has the responsibility of providing space for the structure.

The rp_options structure is declared in the system include file ssu_rp_options.incl.pl1 and has the following format:

```
dcl  1 rp_options                      aligned based (rp_options_ptr),
       2 version                       character (8),
       2 language_info,
         3 non_standard_language       bit (1) aligned,
         3 character_types (0 : 511)
                                       fixed binary (9) unaligned
                                             unsigned,
       2 abbrev_info,
         3 expand_request_lines        bit (1) aligned,
         3 default_profile_ptr         pointer,
         3 profile_ptr                 pointer;
```

STRUCTURE ELEMENTS

version
identifies the version of the rp_options structure being used. The version of the structure described here is given by the value of the named constant RP_OPTIONS_VERSION_1 which is declared in the include file defined above.

language_info
defines the current, default, or new request language depending on which entry point is called.

non_standard_language
for the get entry point, this element is set to "0"b if the standard request language is being used and to "1"b if a nonstandard request language defined by a prior call to the set entry point is in use. For the get_default entry point, this element is always set to "0"b. For the set entry point, a value of "0"b indicates that the caller wishes to use the standard request language; a value of "1"b indicates that the caller wishes to use the nonstandard request language, defined in the character_types array.

character_types
for the get entry point, this element is set to a description of the current request language even if it is the standard language. For the get_default entry point, this element is always set to the description of the standard request language. For the set entry point, this element is only used if non_standard_language above is "1"b and, in this case, is the new definition of the request language. See "Defining a Request Language" below for a description of the contents of this array.

abbrev_info
defines the current, default, or new state of abbreviation processing within this subsystem.

expand_request_lines
for the get entry point, this element is set to "1"b if abbreviations are being expanded in this subsystem; otherwise, it is set to "0"b. For the get_default entry point, this element is always set to "0"b as abbreviation processing is disabled by default. For the set entry point, a value of "1"b indicates that the caller wishes to allow request lines to be expanded; a value of "0"b indicates that the caller wishes to disable abbreviation processing within this subsystem.

default_profile_ptr
for the get entry point, this element is set to locate the default profile segment for abbreviation processing within this subsystem; if the default profile is the default default profile, this element is set to null. For the get_default entry point, this element is always set to null to indicate the default default profile. For the set entry point, the value of this element should be a pointer to the new default profile segment for the subsystem or null if the default default profile is to be used. See "Abbreviation Processing" below for more information.

profile_ptr
for the get entry point, this element is set to locate the current profile segment; if the default profile segment is being used, this element is set to null. For the get_default entry point, this element is always set to null. For the set entry point, the value of this element should be a pointer to the new profile segment to be used within the subsystem or null if the default profile segment is to be used.

Defining a Request Language

A subsystem's request language is defined by specifying the action that the command processor is to take for each possible character appearing on a request line.

The processing type for a character is set in the character_types array defined above in the rp_options structure. The PL/I rank builtin function may be used to access the appropriate entry in the array. For example, to change the processing type of the character "A", a code fragment of the form:

```
rp_options.character_types (rank ("A")) = ...;
```

would be used.

In the following description of the processing types, the term token is used to mean either a request name or an argument to a request.

The possible request language processing types are defined as named constants in the system include file cp_character_types.incl.pl1. The meaning of each of these processing types follows:

NORMAL_CHARACTER
: this character has no special significance to the request processor. It becomes part of whatever token is currently being constructed.

WHITESPACE
: this character separates tokens in the request line but is itself not part of any token. By default, this processing type is used for all the whitespace characters except newline (i.e., space, horizontal tab, vertical tab, and form feed).

COMMAND_SEPARATOR
: this character separates multiple request invocations in the request line and does not become part of any token on the line. By default, this processing type is used for the semicolon (;).

COMMAND_SEPARATOR_OR_WHITESPACE
: this character also separates multiple request invocations in the request line and also does not become part of any token on the line. In addition, when the request processor is rescanning the return value of an active request for tokens (| [...]), this character is treated as a whitespace character. By default, this processing type is used for newline.

SINGLE_TOKEN
: this character separates tokens in the request line. In addition, the character itself becomes a separate token in the request line. For example, if slash (/) is defined as a SINGLE_TOKEN, the request line:

```
list /only tomorrow/
```

contains five tokens: namely, "list" (the request name), slash, "only", "tomorrow", and slash. By default, this processing type is not used for any character.

COMPOUND_TOKEN
: this character separates tokens in the request line. In addition any sequence of characters of this processing type on the request line which appear without intervening characters is treated as a single token. This processing type is useful when defining request languages with multiple character operators. For example, if less-than (<) and equal sign (=) are defined as COMPOUND_TOKEN characters, the request line:

```
print field<= 23
```

contains four tokens: namely, "print", "field", "<=", and "23", while the request line:

```
print field < = 23
```

contains five tokens: namely, "print", "field", "<", "=", and "23" because the less-than and equal signs are separated by another character (a space). By default, this processing type is not used for any characters.

QUOTE_CHARACTER

is used to begin and end a quoted string. Quoted strings do not always correspond to single tokens as explained in Section 3. Only the character which starts the quoted string will terminate it even if several different characters are defined as quote characters. If the character used to start a quoted string is to appear within the quoted string, it must be doubled. For example, if quote (") and apostrophe (') are both defined as quote characters, "list'results" is the quoted string list'results; "list""results" is the quoted string list"results; and, "list"results" is invalid. By default, this processing type is used for the quote (") character.

ACTIVE_STRING_MODIFIER

if this character appears immediately before a begin active string character, it causes the return value to only be scanned for tokens. If two of these characters appear immediately before a begin active string character, it causes the return value to be treated as a single quoted string when constructing tokens on the request line. If this character appears immediately before an end active string character, it causes the return values of iteration within the active string to be treated as if there were no intervening whitespace when forming tokens on the request line. In each of these cases, this character does not appear in any token constructed by the request line processor. If this character appears anywhere else on the request line, it is treated as a normal character. By default, this processing type is used for the vertical-bar (|) character.

BEGIN_ITERATION_1 ... BEGIN_ITERATION_8

this character denotes the beginning of an iteration set. The end of the iteration set is denoted by a character of processing type END_ITERATION_1 if the beginning is denoted by BEGIN_ITERATION_1, by END_ITERATION_2 for BEGIN_ITERATION_2, etc. In other words, there are eight different sets of iteration set delimiters permitted in the request language. Iteration sets correspond to zero or more tokens on the request line as explained in Section 3. The characters which begin the iteration set, however, do not appear as part of any token. By default, the processing type BEGIN_ITERATION_1 is used for left parenthesis and the remaining types are not used.

END_ITERATION_1 ... END_ITERATION_8

this character denotes the end of an iteration set. See BEGIN_ITERATION_1 above for more information. By default, the processing type END_ITERATION_1 is used for right parenthesis and the remaining types are not used.

BEGIN_ACTIVE_STRING_1 ... BEGIN_ACTIVE_STRING_8

this character denotes the beginning of an active string. The end of the active string is denoted by a character of processing type END_ACTIVE_STRING_1 if the beginning is denoted by BEGIN_ACTIVE_STRING_1, by END_ACTIVE_STRING_2 for BEGIN_ACTIVE_STRING_2, etc. In other words, there are eight different sets of active string delimiters permitted in the request language. Active strings correspond to zero or more tokens on the request line as explained in Section 3. The characters which begin and end the active string however, do not appear as part of any token. By default, the processing type BEGIN_ACTIVE_STRING_1 is used for left bracket and the remaining types are not used.

END_ACTIVE_STRING_1 ... END_ACTIVE_STRING_8
this character denotes the end of an active string. See BEGIN_ACTIVE_STRING_1 above for more information. By default, the processing type END_ACTIVE_STRING_1 is used for right bracket and the remaining types are not used.

Abbreviation Processing

The Subsystem Utilities keep track of two profile segments for each subsystem, the default profile and the current profile.

The default profile is the profile segment which is used when abbreviation processing is first enabled within the subsystem and after the user uses the ".u" abbrev request within the subsystem without specifying a profile pathname. If a default profile is not specified by use of the ssu_$set_request_processor_options entry point, the profile segment currently being used at Multics command level is used as the default for the subsystem. This profile segment is known as the default default profile segment.

The current profile is the profile actually being used to expand request line abbreviations and may be changed by the user either by using the ".u" abbrev request or by using the standard subsystem request "abbrev" if it is available within the subsystem.

If a subsystem permits the user to request abbreviation processing through the abbrev request, the command procedure of the subsystem should implement the following control arguments to allow the user to specify the initial state of abbreviation processing:

-abbrev, -ab
enables abbreviation processing within the subsystem. If a default profile is not specified by the -profile control argument, the subsystem will use the same profile being used to expand abbreviations at Multics command level.

-no_abbrev, -nab
disables abbreviation processing within the subsystem. This is the default.

-profile PATH, -pf PATH
specifies the pathname of the default profile to be used within the subsystem. The suffix "profile" is added to PATH if necessary. This control argument implies -abbrev.

As implied above, the profile segment specified on the command line should be used as the default profile segment within the subsystem and not just as the initial current profile. Many users use separate profiles for Multics command level and the various subsystems they use; they then define Multics abbreviations to invoke each subsystem specifying that subystem's profile segment. By having the command procedure make this profile the default for the subsystem, the user can easily switch profiles within the subsystem and then switch back to what they expect to be the default profile by using ".u" without a profile pathname.

WRITING SUBSYSTEM REQUESTS

Writing subsystem requests closely parallels the writing of Multics commands, with the normal concerns for validation, argument processing, and error handling. The parallel extends to the notion of an active function equivalent: a request can be an active request invoked to return a value to be substituted into the request line before normal request line execution.

Whereas valid commands are determined by the user's search rules, the validity of requests is established by a table lookup operation. Before a request can be executed, the request processor must verify that request's presence in a request table. Request tables are defined as described under "Defining Request Tables" below.

The structure of a request is much like that of a Multics command/active function: determine whether or not the request is active; if active, determine where the value is to be returned; determine how many arguments the request is called with; process the arguments and control arguments, making sure they are correct. If everything is determined to be acceptable, the request then performs its defined task, returning a value if it is an active request. A flag setting in the request table indicates how a request should be interpreted (i.e., command request, active request, or both). Since the writer of the request is presumably also defining the request table entry, it is highly unlikely that the request would be called improperly.

Multics commands and subsystem requests can also overlap. A special apply request can cause an arbitrary Multics command line to process an object being manipulated by the subsystem, and requests can be written that also function as Multics commands. All aspects of request writing are examined below.

Argument Processing

A request is always called by the request processor with two parameters, the sci_ptr and the info_ptr, as follows:

```
dcl rq_procedure entry (ptr, ptr);
call rq_procedure (sci_ptr, info_ptr);
```

Argument processing within a request procedure closely resembles command level argument processing. A request that cannot be used as an active request calls ssu_$arg_count to determine the number of arguments, just as a command calls cu_$arg_count. Unlike cu_$arg_count, however, ssu_$arg_count has no error code argument; if invoked by an active request, it calls ssu_$abort_line with the appropriate error code. If a request is written as an active request, or as both an active request and a command request, a call is issued to ssu_$return_arg (equivalent to cu_$af_return_arg) to determine the number of arguments, whether the request is active, and if so, where the value is to be returned (pointer and maximum length). To retrieve any particular argument, a request calls ssu_$arg_ptr (equivalent to cu_$arg_ptr) with the appropriate argument and is returned a pointer and length. Again, unlike cu_$arg_ptr, ssu_$arg_ptr does not return an error code; if asked for a nonexistent argument, ssu_$arg_ptr invokes ssu_$abort_line with the appropriate error code (Expected argument missing).

Error Handling

The subsystem writer must anticipate errors in processing arguments and in performing the actual work of the request. Error handling, like all aspects of writing requests, is similar to the same activity at command level. There are three entry points involved in error handling: ssu_$print_message, ssu_$abort_line, and ssu_$abort_subsystem. Calls to these entry points, much like calls to com_err_ and active_func_err_ at command level, result in messages of the form:

subsystem_name (request_name): status code message user-defined message

If no request name appears in the message, the call was made when no request was being executed (i.e., called by the command procedure, listener, or request processor itself). Status code message is the error message associated with the status code; the user-defined portion is derived from an ioa_ control string.

A call to ssu_$print_message causes a message to be printed followed by a return to the caller (the same as com_err_). This call is used only to print informational or warning messages. The ssu_$abort_line entry point is issued from a request procedure, when to continue executing the request might render totally unexpected results. In this case, a message is printed identifying the cause of termination of the request, and the request line is aborted. In the case of ssu_$abort_subsystem, when the current invocation of the subsystem is to be terminated, a message (if any) is printed, the request line is aborted, the request loop is exited, and the caller is returned to the command procedure.

Whenever calls to ssu_$abort_line and ssu_$abort_subsystem are made, the utilities automatically invoke all cleanup handlers established by the request and any procedures that may have been called which are still active.

The Apply Request

The concept of an apply request is supported in some form in most Multics interactive subsystems (e.g., in send_mail, an apply request can be specified to invoke the emacs editor to edit the message being sent). The ssu_$apply_request_util entry point may be used to simplify the construction of the apply request within a subsystem.

The apply request is used to cause an arbitrary Multics command line to process an object being manipulated by the subsystem. The recommended names for this request are apply and ap; the suggested syntax is:

```
apply {args} {-control_args} command_line
```

1. args
   are optional and denote the object (e.g., message specifiers in read_mail)

2. -control_args
   are optional and either further refine the object (e.g., -header/-no_header in send_mail) or indicate pre- or post-processing to be done to the object (e.g., message filling in send_mail)

3. command_line
   consists of whatever remains of the line after the last -control_arg

The ssu_$apply_request_util entry point constructs the command line, appends to it the pathname of a temporary segment containing the object, and invokes the Multics command processor. The caller of ssu_$apply_request_util (the subsystem's apply request) must parse its arguments to determine where the command line starts on the request line and pass this information to ssu_$apply_request_util. The caller must then create a temporary segment (the recommended method is to call ssu_$get_temp_segment) in which it places the object to be manipulated as determined from its arguments and control arguments. A pointer to this segment and its character count is passed to ssu_$apply_request_util, which returns the character count as updated by the Multics command it executes.

Subsystem Requests and Multics Commands

An additional consideration while writing subsystem requests involves a duality of purpose; i.e., subsystem request as Multics command. The concept takes two forms: a subsystem request that would be useful to invoke as a Multics command, and an existing Multics command/active function that would be useful to employ as a request within the subsystem. The first method (subsystem request as new Multics command) is implemented through a stand-alone invocation; the second (existing Multics command as subsystem request), through the multics_request macro (see "Defining Request Tables" below). Each method is discussed below.

A stand-alone invocation is the mechanism used to write a program that functions both as a subsystem request and as a Multics command. The program as written resembles a subsystem request. When the request is invoked as a request, normal request processing takes place. When the request is invoked as a command, however, a stand-alone invocation must first be created by a call to ssu_$standalone_invocation. This call generates a subsystem invocation in which calls to ssu_ entry points are translated into the respective command-level calls. Calls to ssu_$execute_line and ssu_$evaluate_active_string become calls to cu_$cp and cu_$evaluate_active_string. In the event of error handling, calls to ssu_$print_message, ssu_$abort_line, and ssu_$abort_subsystem become calls to com_err_ or active_fnc_err_ as appropriate. To complete the simulation when ssu_$abort_line or ssu_$abort_subsystem is called, the error message is printed, and then a call is made to an abort procedure supplied by the writer (see ssu_$standalone_invocation) to effect a nonlocal goto back to a point in the program where housekeeping can be performed. Within this sequence, call translation is completely transparent to the subsystem writer.

Depending on how the subsystem writer structures the program, there are potentially three places where the command invocation must be distinguished from the request invocation:

- Upon entry, to create a stand-alone invocation
- Upon normal exit, to destroy the stand-alone invocation
- Upon an abort condition, to destroy the stand-alone invocation

The coding extract below illustrates how the program might be structured.

```
forum_list_meetings command/request

forum_list_meetings: proc () options (variable);

   standalone = "1"b;  /*true indicates command;
                        create a stand-alone invocation*/
   call ssu_$standalone_invocation (sci_ptr, "forum_list_meetings",
           "1.6a", cu_$arg_list_ptr (), abort_entry, code);
      /*set up internal structure for standalone info_ptr*/
   goto COMMON;

list_meetings_request:  entry (a_sci_ptr, a_info_ptr);
   standalone = "0"b; /*false indicates request; process
                       normally*/
   sci_ptr  = a_sci_ptr;
   info_ptr = a_info_ptr;
```

```
COMMON:
   call ssu_$return_arg (sci_ptr, arg_count,
            af_sw, rv_ptr, rv_lth);
      /*proceed with request processing*/
   on cleanup call cleanup_forum_list_meetings ();

   /*actual request code*/

RETURN_FROM_LIST_MEETINGS:  /*abort_entry transfers here*/
   call cleanup_list_meetings ();
   return;
cleanup_list_meetings:
      procedure ();
      /*do housekeeping depending on whether execution
      completed normally; e.g., free any temporary segments*/
   if standalone then do;
      /*free internal structure set up for standalone info_ptr*/
      call ssu_$destroy_invocation (sci_ptr);
   end;
   return;
   end cleanup_list_meetings;

abort_entry:  /*called by ssu_$abort_line or ssu_$abort_subsystem
                  after printing error message*/
      procedure ();
      goto RETURN_FROM_LIST_MEETINGS;
         /*do nonlocal goto after abort condition
         to destroy standalone invocation*/
   end abort_entry;
end forum_list_meetings;
```

The multics_request macro (see "Defining Request Tables" below) is the mechanism for using an existing Multics command/active function as a subsystem request. This is a relatively simple method for greatly expanding a subsystem's request repertoire. In fact, the only commands/active functions that cannot be used in this fashion are those that call either cu_$cp or cu_$evaluate_active_string, in which case they must be implemented using the stand-alone invocation mechanism if they are to be used as both Multics commands and subsystem requests.

Use of the multics_request macro to add existing Multics active functions as subsystem requests is particularly recommended if these active functions would be used frequently with subsystem exec_coms or abbrevs. By allowing these active functions to be invoked directly, many calls to the execute active request may be eliminated, which greatly simplifies exec_com/abbrev writing.

When the request processor is called to process a request, it validates the request definition in the request table. If the definition indicates that the request is, in fact, a Multics command or active function, the request processor calls it as if it were a command or active function. It also translates any command-level error handling calls (com_err_ and active_fnc_err_) to their ssu_ equivalent entry points for proper handling. All of this activity is transparent to the subsystem writer, whose only responsibility in this sequence is to provide the correct request table definition using the multics_request macro.

SUBSYSTEM AREAS AND TEMPORARY SEGMENTS

Each subsystem invocation has associated with it a set of areas and a set of temporary segments from which to acquire temporary storage during that invocation. A call to ssu_$get_area is translated to a call to define_area_ and obtains an area in a temporary segment for subsystem use. A call to ssu_$get_temp_segment is translated to a call to get_temp_segment_ and obtains a temporary segment for subsystem use.

The subsystem writer has the option of acquiring temporary storage through the standard mechanisms (define_area_ and get_temp_segment_) or through the ssu_ entry points. The difference is that any temporary storage acquired through the standard mechanisms has to be released explicitly, while that acquired through calls to ssu_ is released automatically when the invocation is destroyed. If an area acquired through ssu_$get_area is to be released explicitly, it must be released by calling ssu_$release_area. A temporary segment acquired through ssu_$get_temp_segment that is to be released explicitly must be released by calling ssu_$release_temp_segment.

USING EXEC_COMS IN SUBSYSTEMS

The exec_com language (see the Commands manual) can be used to create exec_coms (ecs) for use within individual subsystems through the standard exec_com request (see Appendix J). It should be noted that subsystem ecs execute request lines rather than command lines and pass input to request lines rather than to command lines. Also, the bracket/ampersand constructs evaluate active requests rather than Multics active functions. There are other differences between subsystem ec use and ec use at Multics command level, notably, in the formation of the ec suffix and the ec search list, and in satisfying the referencing_dir rule. These other differences are discussed below.

Subsystem ecs take the subsystem name (e.g., .read_mail) as the default suffix to distinguish them from Multics ecs. The default suffix can be altered by calling ssu_$set_ec_suffix. To retrieve the current value of the ec suffix, call ssu_$get_ec_suffix.

Subsystem ecs do not have a search list by default; rather they are located by specifying the relative pathname of the ec. A search list can be established for the subsystem ecs by calling the ssu_$set_ec_search_list entry point. A call to ssu_$get_ec_search_list retrieves the current ec search list. If a search list is established for a subsystem that has a library of ecs, a call to ssu_$set_ec_subsystem_ptr can be used to set a referencing_dir for the search list.

The subsystem exec_com facility also supports a start_up ec to allow users to simplify use of a subsystem. The name of this ec is start_up.ec_suffix and it is invoked by calling ssu_$execute_start_up, which searches for the start_up ec first in the home directory, then in the project directory, and finally in >site. If no start up is found, subsystem processing proceeds normally. The call to ssu_$execute_start_up should be made before calling the listener (ssu_$listen) and after setting the exec_com suffix, search list, and referencing_dir.

If the subsystem is to support start_up ecs, it is recommended that the command procedure support the following control arguments:

-start_up, -su
: invokes the subsystem's start_up ec, if present. The user's home directory, project directory, and >site are searched in that order. This is the default.

-no_start_up, -nsu, -ns
: does not invoke the subsystem's start_up ec.

TAILORING THE SUBSYSTEM ENVIRONMENT

The subsystem utilities provide a fundamental basis on which to develop and implement interactive subsystems. In some cases, the facilities provided exceed the requirements of the application, and yet other applications may require, to a greater or lesser extent, that the basic facilities be expanded to include additional processing. For this reason, it is possible to tailor parts of the utilities to implement user-coded procedures, or otherwise to override ssu_ default conditions. The prospect of tailoring was addressed earlier, in the discussion on the request language, in which elements of the language could be changed or certain features enabled or disabled (see "Subsystem Request Language" above). The discussion below centers on what are referred to as replaceable procedures, specific pieces of the utilities that the subsystem writer wishes to alter to do something different from the standard interfaces. The following entry points are used to manipulate replaceable procedures:

- ssu_$get_procedure retrieves the current value of the specified replaceable procedure.
- ssu_$set_procedure sets the current value of the replaceable procedure.
- ssu_$reset_procedure resets the specified replaceable procedure to its default value.
- ssu_$get_default_procedure retrieves the default value for the specified replaceable procedure.

A common use of replaceable procedures is to do something in addition to the standard procedure; e.g., each request line, before it is executed, is to be recorded in a segment as a log of subsystem activity. To do this, a call to ssu_$get_procedure retrieves the current execute_line procedure, which is stored in the subsystem's info structure. A call to ssu_$set_procedure sets the new procedure which is invoked whenever a call is made to ssu_$execute_line. This procedure records the request line and then invokes the previous execute_line procedure (saved in the info structure) to do the actual request line execution.

There are 21 replaceable procedures. The two most prominent involve the escape convention and dealing with unknown requests. Four others modify the request loop without completely replacing the listener. The final fifteen are entry points that perform standard operations. Each of these groups is discussed below.

Replaceable Procedures for cpescape and unknown_request

Two replaceable procedures involve processing of the escape convention to Multics command level and dealing with unknown requests.

When reading request lines, the standard subsystem listener interprets any line beginning with ".." as an escape to Multics command level. Such request lines are handled by invoking the cpescape replaceable procedure.

The cpescape replaceable procedure is used as follows:

```
dcl cpescape entry (ptr, ptr, fixed bin(21), fixed bin(35));

call cpescape (sci_ptr, line_ptr, line_lth, code);
```

STRUCTURE ELEMENTS

sci_ptr
is a pointer to the subsystem's control structure as returned by ssu_$create_invocation. (Input)

line_ptr
is a pointer to the request line excluding the leading "..". (Input)

line_lth
is the length of the request line excluding the leading "..". (Input)

code
is a standard system status code. (Output) The listener assumes that the cpescape procedure always prints any necessary error message itself.

By default, the value of the cpescape procedure is ssu_$cpescape, which executes the request line by calling the Multics command processor via cu_$cp. If it is necessary to disable the command processor escape mechanism, the cpescape procedure should be set to ssu_$cpescape_disabled, which prints an appropriate message.

The standard request processor handles unknown request names by calling the unknown_request replaceable procedure. The default value of this procedure prints the standard :

```
Unknown request <request_name>. Type "?" for a request list.
```

message and aborts the request line.

The unknown_request replaceable procedure (which may be invoked directly as ssu_$unknown_request) is used as follows:

```
dcl unknown_request entry (ptr, ptr, char(*), ptr, bit(1)
      aligned);
```

```
call unknown_request (sci_ptr, info_ptr, request_name,
     arg_list_ptr, continue_sw);
```

STRUCTURE ELEMENTS

sci_ptr
is a pointer to the subsystem's control structure as returned by a call to ssu_$create_invocation. (Input)

info_ptr
is a pointer to the subsystem-specific info structure used by this subsystem. (Input)

request_name
is the name of the request as entered on the request line by the user. (Input)

arg_list_ptr
is a pointer to the argument list containing the arguments to be supplied to the request. (Input)

continue_sw
if set to "1"b, indicates that execution of the request line is to continue with the next request. (Input) If set to "0"b, the request processor prints the standard unknown request error message and aborts the request line without further execution.

Request Loop Replaceable Procedures

There are four procedures called by the listener that, by default, do not perform any function. The four procedures, pre_request_line, post_request_line, program_interrupt, and ready, are all called with one argument, the sci_ptr, and may be replaced by user-coded procedures that perform some specific service at that juncture in the request loop.

The pre_request_line procedure is called just before the listener prompts for and reads a request line. A prompt too complex to specify as an ioa_ control string (see "Subsystem Request Loop" above), could be used in a subsystem by replacing this procedure.

The post_request_line procedure is called immediately after successful execution of a request line and, by default, returns to the listener. This provides an opportunity to take some action relative to successful request line execution, before resuming the request loop.

The program_interrupt procedure is called by the listener after it receives a program_interrupt signal, which normally puts the user back at request level. A tailored procedure might, for example, return with a question or simply abort the subsystem.

The ready procedure is called before the pre_request_line procedure, only if ready processing is enabled and the previous request line was nonblank (see "Subsystem Request Loop" above).

Other Replaceable Procedures

The 14 replaceable procedures that match specific entry points are listed below together with the entry point in ssu_ by which each is invoked. The calling sequence of the replaceable procedure is the same as its respective entry point.

Procedure	Entry Point
abort_line	ssu_$abort_line
abort_subsystem	ssu_$abort_subsystem
print_message	ssu_$print_message
listen	ssu_$listen
execute_line	ssu_$execute_line
evaluate_active_string	ssu_$evaluate_active_string
arg_count	ssu_$arg_count
arg_ptr	ssu_$arg_ptr
return_arg	ssu_$return_arg
arg_list_ptr	ssu_$arg_list_ptr
get_default_rp_options	ssu_$default_rp_options
set_request_processor_options	ssu_$request_processor_options
get_request_processor_options	ssu_$request_processor_options
reset_request_processor_options	ssu_$reset_request_processor_options
get_subsystem_and_request_name	ssu_$get_subsystem_and_request_name

As with any software tailoring, changes to one procedure may have a ripple effect on other parts of the software. If, for example, the request processor is changed so that requests are invoked differently (a new execute_line procedure), but the standard requests are still to be used, the subsystem writer must ensure that all the standard interfaces (e.g., ssu_$arg_ptr) are in place. Essentially, that means that all affected request processor procedures must be altered to perform their defined functions within the new request processor environment.

Subsystem Documentation Facilities

Several facilities are available with the utilities to make subsystems self-documenting; i.e., users of a subsystem (like Multics users at command level) can ascertain how it works, what requests are available, what these requests accomplish, etc., by using various help facilities.

In particular, four standard requests enable users to seek online information about the subsystem: help, list_help, summarize_requests (?), and list_requests. A brief description of each of these requests is given below; all standard requests are described in detail in Appendix J.

- The help request prints detailed information on a given topic within the subsystem. If no topic is specified, the help request explains the other requests available to obtain information about the subsystem.

- The list_help request displays the names of all available info segments or those matching a given topic.

- The summarize_requests request (normally invoked as ?) prints a multicolumnar list of most requests available in the subsystem.

- The list_requests request displays the names and brief descriptions of most requests available in the subsystem or those matching a given topic.

The requests listed by the summarize_requests request and those listed by default by the list_requests request are controlled by the dont_summarize and dont_list flags specified in the definitions in the request tables used by the subsystem. See "Defining Request Tables" below for more details.

If standard request tables are not used to define requests in the subsystem, the summarize_requests and list_requests requests will not work as described. Additionally, all four requests will work as described only if ssu_$arg_ptr, ssu_$return_arg, ssu_$arg_count, and ssu_$arg_list_ptr procedures either are not replaced or, if replaced, still perform their defined function (see "Tailoring the Subsystem Environment" above).

SUBSYSTEM INFO SEGMENTS AND DIRECTORIES

The help and list_help requests process subsystem info segments. Usually, there is one info segment for each request in the subsystem, the format of which follows that of a typical command info segment (i.e., name, syntax, function, arguments, control arguments, notes). There may also be info segments dedicated to specific topics. For example, in read_mail there is an info segment dedicated to message specifiers.

Info segments are contained within info directories, which are searched when help and list_help are invoked for a given subsystem invocation. Normally, the call to ssu_$create_invocation names the first info directory in a list of directories to be searched within the subsystem invocation. This list may thereafter be manipulated by the entry points described below.

- ssu_$add_info_dir adds a new directory at the specified location in the list of info directories being searched within this subsystem invocation.

- ssu_$delete_info_dir deletes an info directory from the list of directories being searched.

- ssu_$list_info_dirs obtains the list of info directories currently in use by this subsystem invocation.

- ssu_$set_info_dirs establishes a completely new list of info directories to be searched within this subsystem invocation.

The info directories are searched in order, first to last; the search stops when the named info segment is located. If an invalid directory name is encountered, it is flagged as invalid and no longer searched.

Usually all info segments for a given subsystem are contained in one info directory. Sometimes, however, it is desirable to isolate certain info segments in directories, which are made available only when the requests they pertain to are also available within the subsystem invocation. Availability of the info directories is controlled by the entry points described above.

Occasionally, subsystems will share an info directory. In this situation, the help and list_help requests should ignore info segments from other subsystems in that directory. This is done by establishing an info segment prefix for the subsystem by a call to ssu_$set_info_prefix. The help and list_help requests will recognize only those segment names that begin with the prefix. If an info segment is to be shared among subsystems, it must be given at least one name with each info prefix in order to be found.

USING THE STANDARD REQUESTS INFO SEGMENTS

Info segments (ssu.REQUEST.info) in the directory >doc>subsystem document the standard requests. Subsystems that use the standard requests with recommended names (strongly advised) should do so by adding the standard requests request table (see "Using Standard Requests" below). The info segs for all requests in the table can be included by adding the standard requests info directory. This is accomplished in the following fashion:

```
dcl ssu_info_directories_$standard_requests char (168) external;
      .
      .
      .
call ssu_$add_info_dir
       (sci_ptr, ssu_info_directories_$standard_requests, 9999, code);
```

This adds the standard requests info directory to the end (9999th position) of the list that is used to search for info segments.

Alternatively, subsystems that add one or more of the standard requests to their own request tables (as opposed to adding the standard requests request table) can create links in their subsystem into directories to the appropriate info segments. For example:

```
link >doc>subsystem>ssu.list_requests.info list_requests.info
add_name list_requests.info lr.info
```

When nonstandard names are used for requests or when more information is required (e.g., the exec_com request does not use the default suffix), the subsystem writer should use the standard info segment as the basis for the modified segment.

Subsystem Debugging Facilities

A debug mode facility is provided with the Subsystem Utilities to assist in debugging interactive subsystems. Debug mode is disabled by default. The ssu_$set_debug_mode entry point sets debug mode on or off within the subsystem. A call to ssu_$get_debug_mode returns the current state of debug mode within the subsystem. In addition, the standard debug_mode request may be defined in the subsystem to allow a user (usually the subsystem writer) to enable or disable debug mode from the request loop.

When debug mode is enabled, all calls to ssu_$abort_line and ssu_$abort_subsystem print the specified message, but then, rather than aborting the request line or subsystem, call cu_$cl to invoke another command level. While at this command level, all the debugging facilities of the system are available to determine why ssu_$abort_line or ssu_$abort_subsystem were called. When debugging is completed, the start command causes the request line or subsystem to be aborted.

Subsystem Request Tables

Requests which are valid for a subsystem are defined by a list of request tables. A single request table contains the definition of one or more requests. A request's definition includes the name(s), the procedure that implements the request, the short description printed by the list_requests request, and a set of flags defining how the request is used and whether it known to the summarize_requests and list_requests requests.

When a subsystem invocation is created, the list of request tables is initialized to contain only the request table specified in the call to ssu_$create_invocation. Additional tables may later be added to the request table list by using ssu_$add_request_table. Tables may be removed from the list by calling ssu_$delete_request_table. A call to ssu_$list_request_tables returns the list currently in use, and a call to ssu_$set_request_tables completely replaces the current list.

A request table is a data structure in an object segment created by the ALM assembler (see the Commands manual). Multiple request tables may be defined in a single object segment. Request tables are referenced in calls to the entry points described above by a pointer to the table, which may be constructed as follows:

```
dcl subsystem_tables$default_requests bit (36) aligned external;
   ... addr (subsystem_tables$default_requests) ...
```

The above example constructs a pointer to the default_requests request table in the segment subsystem_tables.

When the request processor looks up the definition of a request, it scans the request table list linearly until it finds the first table containing a definition for a request. This enables, for example, the subsystem writer to provide alternate definitions for standard requests (see below) by placing the supplied ssu_request_tables_$standard_requests request table last in the subsystem's list.

To promote consistency across subsystems, a set of standard requests is supplied with the Subsystem Utilities. These standard request definitions are accessed in a request table by specifying ssu_requests_$<request_name> as the procedure which implements a request, where <request_name> is the name of one of the standard requests described in Appendix J. In addition, the utilities provide a set of standard request tables which may be added to a subsystem's request table list to access specific groups of standard requests. Such a request table would be referenced as:

```
dcl ssu_request_tables_$<table_name> bit (36) aligned external;
```

where <table_name> is the name of one of the standard request tables listed in Appendix J.

Using Standard Requests

Of the standard requests listed in Appendix J, only the summarize_requests and list_requests requests are dependent on the internal format of the Subsystem Utilities data structure. The other standard requests perform as expected provided the ssu_$arg_ptr, ssu_$return_arg, ssu_$arg_count, ssu_$arg_list_ptr, and ssu_$execute_line procedures either are not replaced or, if replaced, still perform their defined function (see "Tailoring the Subsystem Environment" above).

The recommended method of enabling the standard requests within a subsystem is to include the ssu_requests_tables_$standard_requests request table as the last table in the subsystem's request table list. This is accomplished in the following fashion:

```
dcl ssu_request_tables_$standard_requests bit(36) aligned external;
                .
                .
                .
call ssu_$add_request_table (sci_ptr, addr
      (ssu_request_tables_$standard_requests), 9999, code);
```

Those requests that are unsuitable for the subsystem application should be disabled in a earlier request table through use of the unknown_request macro (see below). Those requests with a default behavior that is unsuitable to the subsystem should be replaced by subsystem-specific requests in an earlier table.

In particular, two standard requests, self_identify and quit, are normally replaced by subsystem-specific requests. The standard self_identify request simply prints the subsystem name, version, invocation level, and state of abbreviation processing; it may be desirable to print other information of interest (e.g., the pathname of the mailbox in read_mail). The information printed by the standard request is available through calls to ssu_$get_subsystem_name, ssu_$get_subsystem_version, ssu_$get_invocation_count, and ssu_$get_request_processor_options.

The standard quit request simply exits the subsystem through a call to ssu_$abort_subsystem. It may be desirable through a subsystem-specific request to require permission to exit based on the state of the subsystem or to perform any required housekeeping before exiting (this can also be accomplished by the command procedure).

DEFINING REQUEST TABLES

As described earlier, a request table is a data structure contained in an object segment created by the ALM assembler. A table is defined by the begin_table and end_table macros. Individual requests are defined by the request, unknown_request, and multics_request macros. All of these macros are defined in the ssu_request_macros.incl.alm system include file.

The basic format of the source segment defining one or more request tables is:

```
name      object_segment_name
include   ssu_request_macros
begin_table   table_1
   request 1
      ...
   request n
end_table     table_1
      ...
begin_table   table_N
      ...
end_table     table_N
end
```

where object_segment_name is the name of the object segment and table_i are the names of the request tables. The source segment should be named object_segment_name.alm so that ALM creates the object segment with the desired name.

USING THE REQUEST MACROS

Individual requests are defined by the request, unknown request, and multics_request macros.

The request and multics_request macros contain a set of flags (keywords) that define how the request is to be used. The set_default_flags macro and the set_default_multics_flags macro are available to set the default values for the flags.

Additionally, the set_default_multics_doc macro can be used to supply a documentation string for a subsequent use of the multics_request macro for which there is no explicit documention string.

Syntax

The following rules of syntax apply in the use of these macros. For a complete description of ALM syntax, see the alm command in the *Multics Commands and Active Functions* manual, Order No. AG92.

- Braces indicate that a parameter is optional. Where parentheses appear within braces, the parentheses must be specified as part of the parameter value.
- Whitespace is allowed only before and after the macro name, at the beginning of continuation lines, and within documentation strings.
- A statement may be coded on more than one line by splitting the statement immediately after the comma that separates parameters.
- All parameters are positional, so that commas and parentheses must be specified for parameters to be omitted if later parameters are to be specified.

The request Macro

The request macro is used to define most requests the subsystem writer may wish to include in a request table. The syntax is:

```
request  name,procedure,{(other_names)},{(documentation)},
                                {(system_flags)}
```

where:

1. name
   is the primary name of the request. This name is used in any error messages caused by the request. This parameter is required.

2. procedure
   is the name of an external procedure of two arguments that implements this request. The name must be specified either as refname$entryname or as refname (equivalent to refname$refname). This parameter is required.

3. other_names
   are additional names, separated by commas and enclosed in parentheses, by which this request may be invoked.

4. documentation
   is the brief description of the request printed by the list_requests request. If no documentation string is specified, the request is not listed by the list_request request, unless the -all control argument is specified, in which case the request is listed without any description. The prescribed method for not listing the request is to specify flags.dont_list (see below).

5. system_flags

indicate how the request may be used and how it is documented. The valid flags are:

default
: if specified, means the request is defined with the default flags as set by the last invocation of the set_default_flags macro.

flags.allow_command
: if specified, means the request may be invoked as a command request. This flag is incompatible with flags.allow_both. This is the default.

flags.allow_af
: if specified, means the request may be invoked as an active request. This flag is incompatible with flags.allow_both.

flags.allow_both
: if specified, means the request may be invoked both as a command request and an active request. This flag is incompatible with flags.allow_command and flags.allow_af.

flags.unimplemented
: if specified, means space is reserved in the table for this request, but any attempt to execute it will be rejected. The request is, however, listed by the list_requests request.

flags.dont_list
: if specified, means this request is not to be listed by the list_requests request (unless the -all control argument is specified). The default is to list the request.

flags.dont_summarize
: if specified, means this request is not to be shown by the summarize_requests (?) request. The default is to show the request.

If any flags are specified, the default flags are not assumed and must be explicitly specified by the "default" flag. To specify multiple flags, use a comma-separated list (flagA, flagB....).

Consider the following examples of request definitions from read_mail.

```
request print,rdm_msg_requests_$print_request,
        (pr,p),(Prints the selected messages)
```

This example defines the print request, which is processed by the external procedure rdm_msg_requests_$print_request. The request can optionally be invoked by specifying pr or p. The description "Prints the selected messages" is shown when the print request is listed by the list_requests request. By default, the request can only be invoked as a command request, and is shown by both the list_requests and summarize_requests requests.

```
request list,rdm_msg_requests_$list_request,(ls),
        (List the specified messages),
        (flags.allow_both)
```

This example defines the list request, which is processed by the external procedure rdm_msg_requests_$list_request. The request can optionally be invoked by specifying ls. The description "List the specified messages" is shown when the list request is listed by the list_requests request. The flags.allow_both flag is specified so that the request can be invoked both as a command request and as an active request.

The set_default_flags Macro

The set_default_flags macro can be used to define the default flags for the request macro. The syntax is:

```
set_default_flags (system_flags)
```

where:

1. system_flags
   is a list of one or more of the system_flags as described above for the request macro. Multiple flags must be separated by commas.

Each use of the begin_table macro sets the default flags to (flags.allow_command).

The unknown_request Macro

The unknown_request macro causes the request processor to treat a request as nonexistent, even though it may be defined in a table later in the request table list. This facility is useful for disabling unwanted standard requests specified in the ssu_request_tables_$standard_requests table. Note that in order to insure the request is completely unknown, *all* names (alternate names and short names) must be explicitly disabled. The syntax is:

```
unknown_request name,{(other_names)}
```

where:

1. name
   is the primary name of this request. This parameter is required.

2. other_names
   are additional names, separated by commas and enclosed in parentheses, by which this request may be invoked.

For example, to disable the debug_mode and ready standard requests, the subsystem writer would include the following statements in the definition of a request table that appears before ssu_request_tables_$standard_requests in the subsystem's request table list.

```
unknown_request debug_mode
unknown_request ready,(rdy)
```

The multics_request Macro

The multics_request macro is one of the facilities for defining Multics commands and active functions as subsystem requests (as described earlier under "Subsystem Requests and Multics Commands"). The syntax is:

```
multics_request name,{(other_names)},{(documentation)},
   {procedure},{(system_flags)}
```

where:

1. name
   is the primary name of the request. This name is used in any error messages produced by the command/active function. This parameter is required.

2. other_names
   are additional names, separated by commas and enclosed in parentheses, by which the request may be invoked.

3. documentation
   is the brief description of the request printed by the list_requests request. If no value is specified for this parameter, it defaults to the value set by the last set_default_multics_doc macro within the source segment. If no previous set_default_multics_doc macro was used, the request is not shown by the list_requests request, unless the -all control argument is specified, in which case the request is shown without a description. The set_default_multics_doc macro is described below.

4. procedure
   identifies the Multics command/active function that implements this request. The procedure must be specified as either refname$entryname or refname (equivalent to refname$refname). If omitted, the procedure defaults to the same name as the request (name$name).

5. system_flags
   indicates how the request may be used and how it is documented. Valid flags are described above for the request macro.

The names by which a command/active function are known in a subsystem do not have to be the same as those used at Multics command level.

The following code fragment exemplifies use of the multics_request macro:

```
multics_request date_time_equal,(dteq),(Compare date/time strings),,
     (default,flags.dont_list,flags.dont_summarize)
```

This example defines the Multics date_time_equal command/active function as a subsystem request of the same name, with a short name of dteq. The description "Compare date/time strings" is printed when the request is listed by the list_requests request. Note that, since flags are specified, a comma denotes the missing parameter, procedure, which is unnecessary because the request name is the same as the command/active function.

The set_default_multics_flags Macro

The set_default_multics_flags macro can be used to define the default flags for the multics_request macro. The syntax is:

```
set_default_multics_flags (system_flags)
```

where:

1. system_flags
   is a list of one or more of the system_flags as described above for the request macro. Multiple flags must be separated by commas.

Each use of the begin_table macro sets the default flags to (flags.allow_command, flags.allow_af).

The set_default_multics_doc Macro

The set_default_multics_doc macro may be used to supply a documentation string for any subsequent use of the multics_request macro for which there is no explicit documentation string. The syntax is:

```
set_default_multics_doc (documentation)
```

where:

1. documentation
   is the new default string for use by subsequent multics_request macros. Within the documentation string, &1 is replaced by the name of the request being defined. For example:

```
set_default_multics_doc (Type '..help &1' for more information.)
```

SECTION 5

INPUT AND OUTPUT FACILITIES

This section contains information on the various input and output facilities available on the Multics system. A general description of the input/output (I/O) system is contained in "Multics Input/Output System" below. The section also contains information on programming language I/O, file I/O, terminal I/O, bulk I/O, and how to implement user-written I/O modules. In addition, Multics peripheral I/O facilities including the Resouce Control Package (RCP) and I/O Interface (IOI) are described from the user's viewpoint.

Earlier versions of Multics used a different, but similar, I/O system. Parts of the system documentation may still use the terminology of the old I/O system. In particular, the old system used the term "I/O stream" instead of "I/O switch" and the terms "DIM" and "IOSIM" instead of "I/O module." Also, documentation may describe attaching to a device even though the attachment may be to something other than a device, e.g., a file in the storage system. (A file is defined as a segment or multisegment file.)

MULTICS INPUT/OUTPUT SYSTEM

Since the Multics input/output (I/O) system handles logical I/O rather than hardware I/O, I/O on the Multics system is essentially device independent. Most I/O operations refer only to logical properties (e.g., the next record, the number of characters in a line) rather than to particular device characteristics or file formats. The system permits I/O to and from files in the storage system. This involves only the transfer of data from one memory location to another. It does not deal with the transfer of pages (paging) between secondary storage and main memory. This paging is managed invisibly by the Multics virtual memory and is used by user programs and the I/O system alike. Hardware I/O is performed by routines that are not normally called by a user.

To facilitate control of the sources and targets for I/O, the system makes use of a software construction called an I/O switch. An I/O switch is like a channel in that it controls the flow of data between program accessible storage and devices, files, etc. The switch must be attached before it can be used. The attachment specifies the source/target for I/O operations and the particular I/O module that performs the operations. For example, a switch may be attached to the user's terminal through the tty_ I/O module or to a file in the storage system through the vfile_ I/O module. The basic tool for making attachments and performing I/O operations is the iox_ subroutine (described in the Subroutines manual). All functions of the I/O System are accessible through calls to this subroutine.

Attachments and I/O operations can also be done from command level, using the io_call command. The print_attach_table command prints descriptions of all current attachments. Both of these commands are described in the Commands manual.

System Input/Output Modules

The Multics system contains the following I/O modules, which, unless otherwise noted, are described in the Subroutines manual:

audit_
: provides a mechanism for auditing and editing I/O on a switch.

bisync_
: performs stream I/O over a binary synchronous communications channel.

cross_ring_
: allows an outer ring to attach a switch to a preexisting switch in an inner ring to perform I/O operations.

discard_
: is a sink for unwanted output.

g115_
: performs stream I/O from/to a Honeywell Level 6 G115 data transmission terminal.

hasp_host_
: simulates record-oriented I/O to a single device of a workstation while communicating with a host system using the HASP communications protocol.

hasp_workstation_
: performs record-oriented I/O to a single device of a remote terminal that supports the HASP communications protocol.

ibm2780_
: performs stream I/O from/to a device similar to the IBM 2780 data transmission terminal.

ibm3270_
: performs stream I/O from/to a device similar to the IBM 3270 data transmission terminal.

ibm3780_
: performs stream I/O from/to a device similar to the IBM 3780 data transmission terminal.

mtape_
: supports I/O to/from tapes written in ANSI and IBM tape formats. This is an extended I/O module which uses opening, closing and detaching descriptions to tailor its operations. (Users writing new applications should use mtape_ instead of tape_ansi_ or tape_ibm_ I/O modules).

rdisk_
: supports I/O from/to removable disk packs.

record_stream_
: provides a mechanism for doing record I/O on an unstructured file and stream I/O on a structured file.

remote_input_
: performs record input from a terminal I/O module that is assumed to be connected to a remote I/O device.

remote_printer_
: formats and controls stream I/O to a remote I/O terminal that has the characteristics of a line printer.

remote_punch_
: formats and controls stream I/O to a remote I/O terminal that has the characteristics of a card punch.

remote_teleprinter_
: formats and controls stream I/O from/to a logical entity that has the characteristics of a teleprinter.

report_
: supports input to the report generation portion of a Multics Report Program Generator (MRPG) object segment *(Multics Report Generator Reference Manual,* Order No. CC69).

signal_io_
: signals a condition whenever an iox_ operation is performed.

syn_
: establishes one switch as a synonym for another.

tape_ansi_
: supports I/O from/to magnetic tape files according to standards established by the American National Standards Institute (ANSI) (Users writing new applications should use the mtape_ I/O module).

tape_ibm_
: supports I/O from/to magnetic tape files according to standards established by IBM (Users writing new applications should use the mtape_ I/O module).

tape_mult_
: supports I/O from/to magnetic tape files in Multics standard tape format.

tape_nstd_
: supports I/O from/to tapes in nonstandard or unknown formats.

tty_
: supports I/O from/to terminals.

vfile_
: supports I/O from/to files in the storage system.

window_io_
: implements a virtual video terminal (a window) on the user's terminal and provides real-time editing of input.

How to Perform Input/Output

To perform I/O, carry out the steps listed below. In general, a step may be performed by a call to the iox_ subroutine (described in the Subroutines manual) or by use of the io_call command (described in the Commands manual). The I/O facilities of programming languages may also be used to carry out these steps.

In steps 2, 4 and 5 below, the I/O switch is opened, closed and detached. Some I/O modules accept file open, close and detach description arguments which allow the user to tailor the operation to his needs. These are called extended I/O modules. Use the iox_$open_file, iox_$close_file and iox_$detach entrypoints for extended I/O modules; and use iox_$open, iox_$close and iox_$detach_iocb for nonextended I/O modules (those which do not accept such descriptions). Currently, the only extended I/O module provided with the Multics system is the mtape_ I/O module.

The I/O facilities of programming languages (eg, PL/I and Fortran) can perform these steps without using iox_ or io_call. Currently, programming language facilities can only interface with nonextended I/O modules.

1. Attach an I/O switch. This step specifies a source/target for subsequent I/O operations and names the I/O module that performs the operations. Example:

   ```
   io_call attach input_sw vfile_ some_file
   ```

   This command line attaches the switch named input_sw to a storage system file whose relative pathname is some_file. The I/O module that performs this operation is named vfile_ (described in the Subroutines manual). This attachment could also have been performed by a subroutine call as follows:

   ```
   call iox_$attach_name ("input_sw", iocb_ptr,
    "vfile_ some_file", codeptr (procedure_name), code);
   ```

   where procedure_name names the external procedure that is calling iox_$attach_name, and codeptr is a Multics-specific PL/I built-in function.

2. Open the I/O switch. This step prepares the switch for a particular mode of processing (e.g., reading records sequentially) using the already established attachment. An example of a nonextended I/O module opening:

```
call iox_$open (iocb_ptr, Sequential_input, "0"b, code);
```

The iocb_ptr identifies the switch (see "Input/Output Switches" below). The argument Sequential_input means that the opening is for sequential reading and is a constant declared in the iox_modes.incl.pl1 include file. The "0"b represents an obsolete argument. See the description of the iox_ subroutine for full details. This openning for a nonextended I/O module could also have been performed by a command, as follows:

```
io_call open input_sw sequential_input
```

For an extended I/O module, the open subroutine call would look like:

```
call iox_$open_file (iocb_ptr, Sequential_input,
 "-name employee_data", "0"b, code);
```

The arguments are the same as for iox_$open except for the third argument, which is the open description. Assuming that the original attachment was to a tape volume, the open description in the example above specifies which file on tape to read by giving the file name. This opening could be performed by a command as follows:

```
io_call open_file input_sw sequential_input
     -name employee_data
```

3. Perform the required data transfer and control I/O operations working through the switch. For example, read one record at a time until an end-of-information code is returned by the read operation. Example:

```
call iox_$read_record (iocb_ptr, buffer_ptr,
 buffer_length, actual_record_length, code);
```

This read_record step could also have been performed by the io_call command.

```
io_call read input_sw
```

The io_call command prints the record which is read.

4. Close the I/O switch. This step cleans up by writing out buffers, marking the end of a file, etc. The I/O switch is restored to the state it was in after step 1. The close could be followed by a repeat of steps 2-4, perhaps with a different opening mode or different open description. An example of a nonextended I/O module opening is:

```
call iox_$close (iocb_ptr, code);
```

Closing of a nonextended I/O module could also have been performed by the io_call command, as follows:

```
io_call close input_sw
```

For an extended I/O module, the close subroutine call would look like:

```
call iox_$close_file (iocb_ptr,
     "-close_position eof", code);
```

The arguments are the same as for iox_$close except for the second argument, which is the close description. Assuming that the original attachment was to a tape volume, the close description in the example above specifies to position the tape to the end of the file being read after the file is closed. This close operation could be performed by a command as follows:

```
io_call close_file input_sw -close_position eof
```

5. Detach the I/O switch. After this step, the switch can be attached again for some other purpose. An example of detaching a nonextended I/O module is:

```
io_call detach input_sw
```

This detachment step could also have been performed by a subroutine call as follows:

```
call iox_$detach_iocb (iocb_ptr, code);
```

For an extended I/O module, the detach operation would look like:

```
io_call detach input_sw -unload
```

The example detach description for a tape volume specifies that the tape volume is to be unloaded from the tape drive as part of the detach operation. An equivalent subroutine call is:

```
call iox_$detach (iocb_ptr, "-unload", code);
```

In general, step 1 (attach) specifies a particular type of device or volume. For nonextended I/O modules, step 1 also identifies file name and file format. For extended I/O modules, file name and format information is given in step 2 (open). It is often convenient to have these steps and step 5 (detach) performed from command level, while other steps are performed by a program. This approach may be used to make a program device independent. Another approach is to include the attach and open calls in the user program, but to have the program prepared to accept the status code from iox_$attach or iox_$open indicating the switch is already attached or opened. The program should detach a switch only if it attached it, and close a switch only if it opened it.

Input/Output Switches

Each I/O switch has an I/O control block (IOCB) associated with it. Storage for the control block is automatically allocated when the switch is attached. The contents of the control block are maintained by the I/O system and are not usually of interest to the general user. It does, however, contain two pointers of interest.

1. iocb.attach_descrip_ptr
   is a pointer to a character string describing the attachment of the switch. If the pointer is null, the switch is not attached.

2. iocb.open_descrip_ptr
   is a pointer to a character string describing the opening mode and optional open description of the switch. If the pointer is null, the switch is not open.

Each I/O switch has a name that is used to refer to the I/O switch at command level and is also used in other contexts where reference by a character string name is appropriate. Most calls to the iox_ subroutine reference an I/O switch by its control block pointer. Given the switch name, the iox_$find_iocb entry point returns the control block pointer. The switch name is a character string from one to 32 characters long with no blanks.

Each I/O switch belongs to a particular ring, normally the user ring. Within a ring, switch names are unique, but switches in different rings may have the same name.

ATTACHING A SWITCH

To attach a switch, the "io_call attach..." command or the iox_$attach_ptr or iox_$attach_name entry points should be invoked. In all cases, an attach description must be given. This string has the following form:

```
module_name options
```

where module_name and each option are separated from one another by one or more blanks. If an option contains blanks it must be enclosed in quotes ("). If an option already contains a quote, the quote must be doubled.

The module_name determines the I/O module for the attachment as follows: If it does not contain any instances of greater than or less than characters (> or <), it is interpreted as a reference name, and the I/O module is found by the search rules. If module_name contains any greater than or less than characters, it is interpreted as the pathname (absolute or relative) of the I/O module.

The options must conform to the requirements of the particular I/O module. The I/O modules are described in the manuals mentioned above in "System Input/Output Modules." In general, the first option listed is the source/target of the attachment (i.e., the name of the device or file).

When the attachment is made, if the I/O module is not already initiated by the specified reference name, it is so initiated. When module_name is given as a pathname, the reference name is the final entryname in the pathname.

The attach description associated with the attached switch (and accessible through the print_attach_table command, described in the Commands manual) may not be exactly the same as the attach description given to the io_call command or the iox_$attach_ptr or iox_$attach_name entry points. In general, the I/O module transforms the attach description into a standard form. For example, the command:

```
io_call attach foo >ldd>sdd>vfile_ my_file
```

might generate the attach description:

```
vfile_ >udd>m>JRDoe>my_file
```

OPENING A SWITCH

The "io_call open ..." command and the iox_$open subroutine are used to open a switch attached through a nonextended I/O module. The "io_call open_file ..." command and iox_$open_file subroutine are used to open a switch attached through an extended I/O module (one which accepts open, close and detach descriptions). In either case, one of the opening modes listed in Table 5-1 must be specified. As shown in Table 5-1, the opening mode determines which I/O operations may be carried out through the open switch. Whether or not opening in a particular mode is possible depends on the attachment of the switch. The relation between opening modes and file attachments is discussed in "File Input/Output" below. For other types of attachments see the description of the particular I/O module. Table 5-2 shows the type of opening modes supported by each I/O module.

An open description can be used with extended I/O modules to complete the specification of the file being opened. For the io_call command interface to open_file, the open description is optional. For the iox_ subroutine interface to $open_file, the open subroutine argument is required, but it can be a null string indicating that no option was given. A sample open description is provided below.

```
-name employee_data -format fb -record 80 -block 800
     -display
```

The control arguments and operands are separated from one another by one or more blanks. If an operand contains blanks, it must be enclosed in quotes ("). If an operand contains a quote, this quote must be doubled and the operand must be enclosed in quotes.

CLOSING A SWITCH

The "io_call close ..." command and iox_$close subroutine are used to close a switch attached through a nonextended I/O module. The "io_call close_file ..." command and iox_$close_file subroutine are used to close a switch attached through an extended I/O module.

A close description can be used with extended I/O modules to specify the disposition of the file being closed. For the io_call command interface to close_file, the close description is optional. For the iox_ subroutine interface to $close_file, the close subroutine argument is required, but it can be a null string indicating that no option was given. A sample close description is provided below.

```
-close_position bof -display -comment "File read complete."
```

The control arguments and operands are separated from one another by one or more blanks. If an operand contains blanks, it must be enclosed in quotes ("). If an operand contains a quote, this quote must be doubled and the operand must be enclosed in quotes.

DETACHING A SWITCH

The "io_call detach_iocb ..." command and iox_$detach_iocb subroutine are used to detach a switch attached through a nonextended I/O module. The "io_call detach ..." command and iox_$detach subroutine are used to detach a switch attached through an extended I/O module.

A detach description can be used with extended I/O modules to specify the disposition of the device or volume being detached. For the io_call command interface to detach_file, the detach description is optional. For the iox_ subroutine interface to $detach, the detach subroutine argument is required, but it can be a null string indicating that no option was given. A sample detach description is provided below.

```
-unload -display -comment "Operator: put this tape in bin 23"
```

The control arguments and operands are separated from one another by one or more blanks. If an operand contains blanks, it must be enclosed in quotes ("). If an operand contains a quote, this quote must be doubled and the operand must be enclosed in quotes.

SYNONYM ATTACHMENTS

By means of the syn_ I/O module, an I/O switch (e.g., switch_1) may be attached as a synonym for another I/O switch (e.g., switch_2). In general, performing an I/O operation through switch_1 then has the same effect as performing it through switch_2. There are two exceptions:

1. Detaching switch_1 simply breaks the synonymization and has no effect on switch_2.

2. The attach description for the synonym attachment may specify that certain operations are to be inhibited. An attempt to perform an inhibited operation through switch_1 results in a status code that indicates an error.

Synonym attachments are especially useful when one wishes to switch the source/target for a set of I/O operations. For example, the I/O switch user_output is normally attached as a synonym for user_i/o (which is normally attached to the user's terminal). The following command lines can be used to create an I/O switch named file_switch and attach it to a file, open file_switch for stream_output, detach the I/O switch user_output, and make the I/O switch user_output a synonym attachment to the I/O switch file_switch.

```
io_call attach file_switch vfile_ file_name -extend
io_call open file_switch stream_output
ready_off
io_call detach user_output
io_call attach user_output syn_ file_switch
```

The result of these five command lines is that output that would normally be sent to a terminal is written into a file. The file_output command (described in the Commands manual) performs this sequence of steps and is the normal way of directing terminal output to a file. Note the presence of the ready_off command. Without this command a fatal process error will occur when the ready message following the io_call detach user_output command attempts to print. This is due to the fact that the switch on which the output should go is no longer attached.

The following command lines can be used to undo the effects of the previous four command lines with the result that subsequent output to the I/O switch user_output is written on the user's terminal. The revert_output command (described in the Commands manual) performs this sequence of steps and is the normal way of reverting user_output to its normal attachment (the terminal).

```
ready_off
io_call detach user_output
io_call attach user_output syn_ user_i/o -inh close
    get_line get_chars
io_call close file_switch
io_call detach file_switch
```

It is possible to have a chain of synonyms; e.g., switch_1 as a synonym for switch_2 and switch_2 as a synonym for switch_3. The final switch in the chain is the actual I/O switch for all the other switches in the chain. More precisely, if an I/O switch, switch_1, is not attached as a synonym, then its associated actual I/O switch is itself. If switch_1 is attached as a synonym for switch_2, then the actual I/O switch associated with switch_1 is the same as the actual I/O switch associated with switch_2.

With the notion of the actual I/O switch, the effect of a synonym attachment of an I/O switch, switch_1, can be precisely described as follows:

1. The open_description of switch_1 is the same as the open_description of the actual I/O switch associated with switch_1. (Hence switch_1 is open or closed according to whether the actual switch is open or closed.)

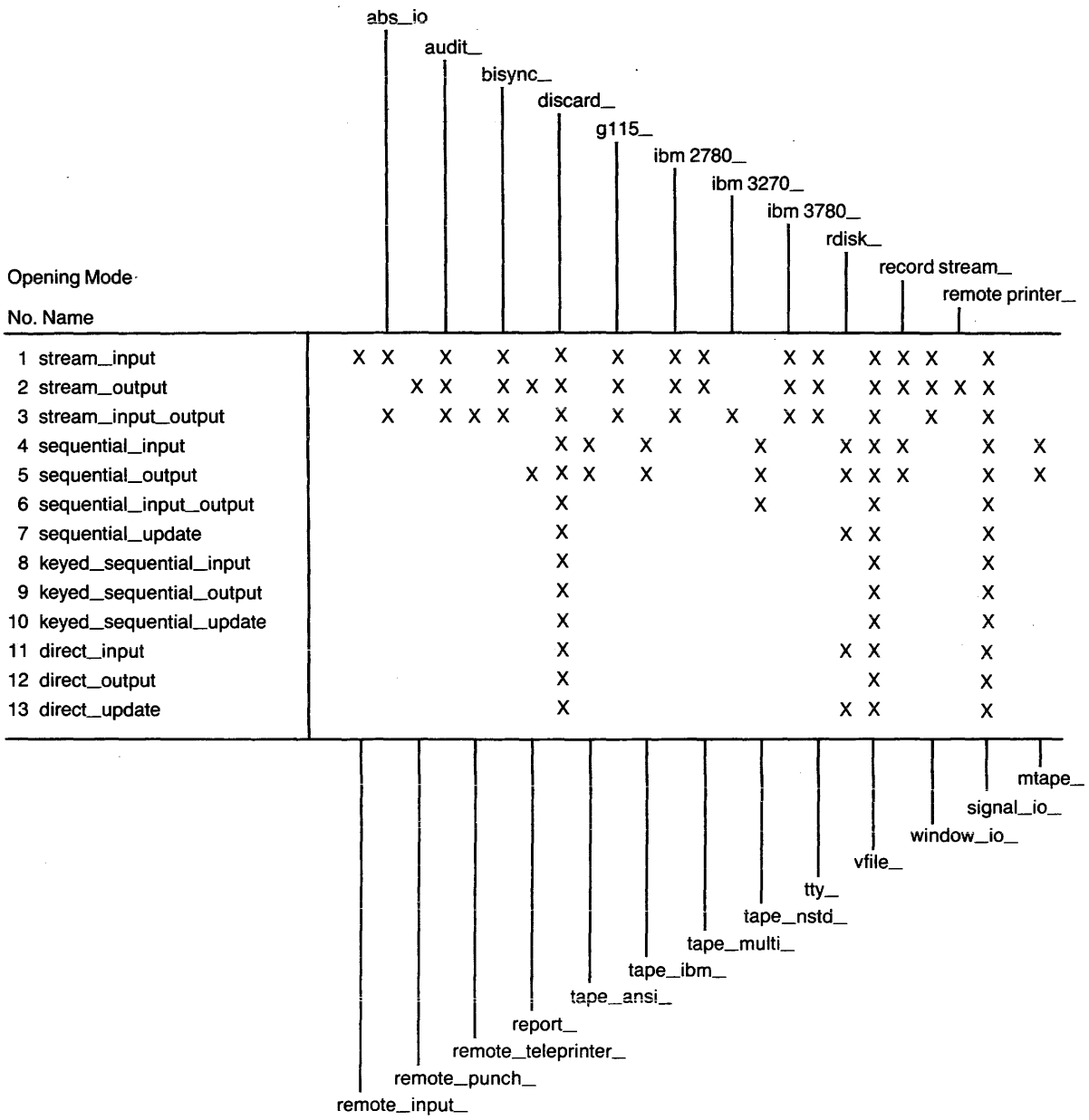
2. If the open I/O operation or one of the I/O operations listed in Table 5-1 is performed through switch_1, then the effect is the same as if it were performed through the actual I/O switch associated with switch_1, with one exception. The exception is that if any synonym attachment in the chain (connecting switch_1 to the actual I/O switch) inhibits the operation, then the only effect is to return a status code that indicates an error.

Table 5-1. Opening Modes and Allowed Input/Output Operations

Opening Mode No.	Name	get_line	get_chars	put_chars	read_record	rewrite_record	delete_record	read_length	position	seek_key	read_key	close or close_file	write_record	control	modes
1	stream_input	X	X						2			X		1	1
2	stream_output			X								X		1	1
3	stream_input_output	X	X	X					2			X		1	1
4	sequential_input				X			X	X			X		1	1
5	sequential_output											X	X	1	1
6	sequential_input_output				X			X	X			X	X	1	1
7	sequential_update				X	X	3	X	X			X	4	1	1
8	keyed_sequential_input				X			X	X	X	X	X		1	1
9	keyed_sequential_output									X		X	X	1	1
10	keyed_sequential_update				X	X	X	X	X	X	X	X	X	1	1
11	direct_input				X			X		X		X		1	1
12	direct_output									X		X	X	1	1
13	direct_update				X	X	X	X		X		X	X	1	1

1. Depends on the attachment.
2. Allowed if attached to a file in the storage system.
3. Allowed unless file is blocked.
4. Allowed for blocked and sequential files in the storage system.

Table 5-2. Opening Modes Supported by I/O Modules



The syn_ I/O module is not included in this table because the allowed modes are a function of the switch to which the syn_ module is being attached.

STANDARD INPUT/OUTPUT SWITCHES

Four I/O switches are attached as part of the standard initialization of a Multics process.

Switch	*Normal Attachment*
user_i/o	the user's terminal
user_input	synonym for user_i/o
user_output	synonym for user_i/o
error_output	synonym for user_i/o

These switches may be attached in other ways, but the user must always attach user_input, user_output, and error_output as synonyms. The attachment of user_io differs for absentee processes and network connections. However, this difference is only significant in that the attachment of user_i/o may not be to the user's terminal.

When the "video" system is activated by a call to video_utils_$turn_on_login_channel or by executing the window_call invoke command, the existing attachments of the terminal are removed and replaced with video system attachments. The I/O switch user_i/o is now attached through the I/O module window_io_ to a new I/O switch, user_terminal_. The user_terminal I/O switch is attached through the I/O module tc_io_ to the terminal.

INITIALIZATION OF EXTERNAL POINTER VARIABLES

The following external pointer variables are initialized to point to the control blocks for the corresponding I/O switches:

```
dcl iox_$user_io external pointer;
dcl iox_$user_input external pointer;
dcl iox_$user_output external pointer;
dcl iox_$error_output external pointer;
```

These variables must never be modified. By using these variables, one can save time and avoid calls to the iox_$find_iocb entry point to locate these commonly used control blocks. Thus, a simple and efficient way to write to

```
call iox_$put_chars (iox_$user_output, buffer_ptr, buffer_length, code);
```

Interrupted Input/Output Operations

It may happen that an I/O operation being performed on a particular I/O switch, switch_1, is interrupted, e.g., by a quit signal or an access violation signal. In general, until the interrupted operation is completed, or until switch_1 is closed, it is an error (with unpredictable consequences) to perform any I/O operation except close on switch_1. However, some I/O modules (tty_ in particular) allow other operations on switch_1 in this situation. (See the Subroutines manual for details.) If the switch switch_1 is closed while the operation is interrupted, control must not be returned to the interrupted operation.

PROGRAMMING LANGUAGE INPUT/OUTPUT FACILITIES

It is possible to perform I/O through a particular switch using both the facilities of a programming language and the facilities of the I/O system (invoked directly). The following statements about this sort of sharing of switches apply in most cases:

1. The I/O system may be used to attach a switch or to attach and open it. The language I/O routines are prepared for this, and they close (detach) a switch only if they opened (attached) it.

2. A switch opened for stream_input may be used both directly and through language I/O if care is exercised. In general, the languages read a line at a time. Thus the order of input may get confused if a direct call is made to the I/O system while the language routines are processing a line. Trouble is most likely to arise after issuing a quit signal (pressing the appropriate key on the terminal, e.g., ATTN, BRK, etc.).

3. A switch opened for stream_output may be used both directly and through language I/O if formatting by column number, line number, page number, etc. is not important. Some shuffling of output may be expected, especially if a direct call to the I/O system (e.g., by the issuing of a quit signal) is made while the language I/O routines are processing an I/O statement.

4. If a switch is opened for record I/O (sequential, keyed_sequential, and direct modes), using it both directly and through language I/O is not recommended.

A direct call to the I/O system has no effect on file information and buffers maintained by the language I/O routines and is likely to cause garbled input or output. The close_file command (described in the Commands manual) closes PL/I, Pascal and FORTRAN file information blocks used by the language I/O routines. For details on the facilities of a particular language and for a discussion of the usage of related Multics commands, see the reference manual and/or user's guide for that language.

While most language I/O facilities can pass a complete attach description to the I/O system, they have no way of passing the open, close and detach descriptions required by extended I/O modules.

FILE INPUT/OUTPUT

The I/O system distinguishes four types of files: unstructured, sequential, blocked, and indexed. These types pertain to the logical structure of a file, not to the file's representation in storage, on magnetic tape, etc. For example, in the storage system a file may be stored as a single segment or as a multisegment file; but this does not affect the meaning of I/O operations on the file.

Unstructured Files

An unstructured file contains a sequence of 9-bit bytes. Normally the bytes are ASCII characters, but this is not required.

The following I/O operations apply to unstructured files:

get_line
: reads a line from the file, i.e., a sequence of bytes ending with an ASCII newline character

get_chars
: reads a specified number of bytes

put_chars
: adds bytes at the end of the file

position
: positions to the beginning or end of the file, skips forward or backward over a specified number of records:

```
-1  goes to the beginning of the file
+1  goes to the end of the file
 0  skips newline characters or records (lines)
 2  positions to an absolute record (line)
 3  skips characters
```

Sequential Files

A sequential file contains a sequence of records. Each record is a string of 9-bit bytes. A record may be zero length.

The following I/O operations apply to sequential files:

read_record
: reads the next record

read_length
: obtains the length of the next record

write_record
: adds a record to the file or replaces a record

rewrite_record
: replaces a record

delete_record
deletes a record

position
positions to the beginning or end of the file, skips forward or backward over a specified number of records:

```
-1  goes to the beginning of the file
+1  goes to the end of the file
 0  skips records
 2  positions to an absolute record
```

Blocked Files

A blocked file contains a sequence of records. Each record is a string of 9-bit bytes. The length of a record may range from zero to a preset maximum value associated with the file.

The following I/O operations apply to blocked files:

read_record
reads the next record

read_length
obtains the length of the next record

write_record
adds a record to the file or replaces a record

rewrite_record
replaces a record

position
positions to the beginning or end of the file, skips forward or backward over a specified number of records:

```
-1  goes to the beginning of the file
+1  goes to the end of the file
 0  skips records
 2  positions to an absolute record
```

Indexed Files

An indexed file contains a sequence of records and an index. Each record is a string of 9-bit bytes. A record may be zero length.

The index associates each record with a key. A key is a string of from 0 to 256 ASCII characters containing no trailing blanks. Ordinarily, no two records in the file have the same key. The order of records in the sequence is key order: record x precedes record y if and only if the key of x is less than the key of y according to the Multics PL/I rules for string comparison (lexicographic order using the ASCII collating sequence).

All the I/O operations applicable to sequential files apply to indexed files as well; however, write_record only adds records. In addition, the following two operations manipulate keys:

read_key
: obtains the key and length of the next record

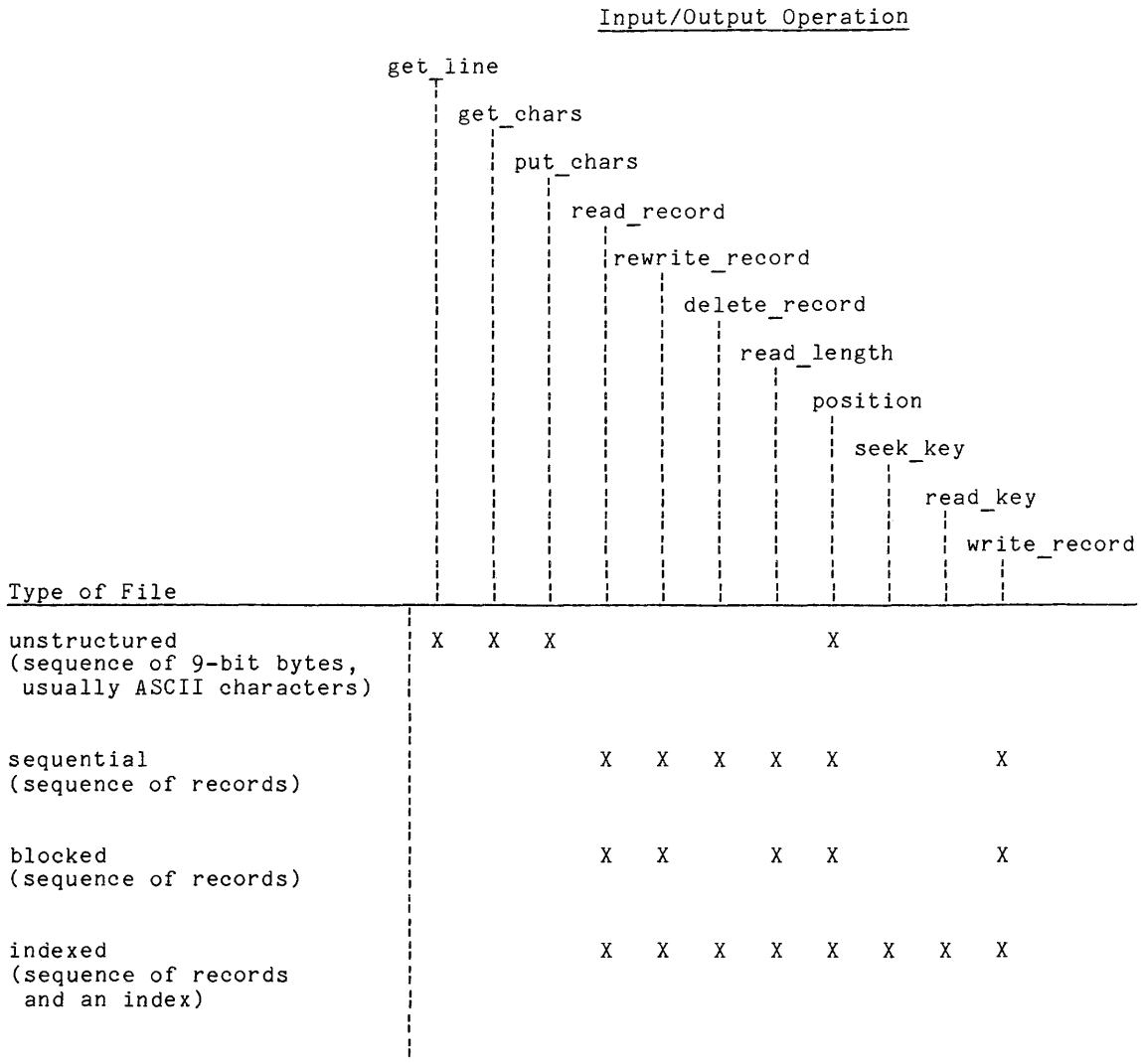
seek_key
: positions to the record with a given key or defines the key to be associated with a record to be added (by a subsequent write operation)

position
: positions to the beginning or end of the file, skips forward or backward over a specified number of records:

```
-1  goes to the beginning of the file
+1  goes to the end of the file
 0  skips records
 2  positions to an absolute record
```

Table 5-3 shows the I/O operations that are permitted with each type of file.

Table 5-3. File Types and Allowed Input/Output Operations



Each record is a string of bytes; a record may be of zero length. A blocked file has a characteristic maximum record length that is initially set by the user. For an indexed file, a key is a string of 0 to 256 ASCII characters, with no trailing blanks.

File Opening

When an I/O switch is attached to a file and is opened for input, the file must exist and must be compatible with the opening mode. Table 5-4 shows the compatibility between file types and opening modes.

When the opening is for output, input_output, or update, and the file does not exist, a file of the appropriate type is created. File creation can be suppressed in storage system files; see the description of the vfile_ I/O module in the Subroutines manual for details. The type of file created by a particular mode of opening is shown in Table 5-4.

When the opening is for output or input_output, and the file already exists, it is normally replaced by an empty file of the appropriate type. However, if the attachment specifies extension of the file, the file is not replaced. In this case the file must be compatible with the opening mode.

For files, opening for input_output means opening with the intent of first writing the file and then reading it during the same opening. An existing file is replaced by an empty file unless extension is specified.

Table 5-4. Compatible File Attachments

Opening Mode		File Type			
No.	Name	unstructured	sequential	blocked	indexed
1	stream_input	x	1	1	1
2	stream_output	x3			
3	stream_input_output	x3			
4	sequential_input		x	x	x
5	sequential_output		x3	x3	
6	sequential_input_output		x3	x3	
7	sequential_update		2,3	x3	x
8	keyed_sequential_input				x
9	keyed_sequential_output				x3
10	keyed_sequential_update				x3
11	direct_input				x
12	direct_output				x3
13	direct_update				x3

1. The structure of the file is ignored and everything in it is treated as data (including control words).

2. The file must be in the storage system.

3. This type of file is created by an output or update opening for the specified mode unless this feature is explicitly suppressed. Update openings never replace an existing file. (See the individual I/O module descriptions in the Subroutines manual to see which control arguments are applicable.)

File Closing

When an I/O switch attached to a file has been opened for output, input_output, or update, a close operation should be performed on the switch before the process is terminated. If not, the file may be left in an inconsistent state; e.g., an end of file mark may not be written for a tape file, or the bit count of a segment may not be set for a storage system file.

When a process terminates due to invocation of the logout or new_proc command, all I/O switches are closed by the epilogue handler for the process. The epilogue handler for a run unit, which is called by the stop_run command, or by normal run unit termination, closes all I/O switches within the run unit.

File Position Designators

The I/O operations on files are defined in terms of four position designators. In cases where several I/O switches are open and attached to the same file, each opening has its own set of designators. The designators are:

next byte
: the first byte to be read by the next get_line or get_chars operation

next record
: the record to be read by the next read_record or inserted by the next write_record operation

current record
: the record to be replaced or deleted by the next rewrite_record or delete_record operation

key for insertion
: the key to be associated with the record added to an indexed file by the next write_record operation

The initial values for these designators are shown in Table 5-5.

Table 5-5. File Position Designators at Open

Opening Mode		Designator(1)			
No.	Name	next byte	next record	current record	key for insertion
1	stream_input	first byte			
2	stream_output	end of file			
3	stream_input_output	end of file(2)			
4	sequential_input		first record		
5	sequential_output				
6	sequential_input_output		end of file(2)		
7	sequential_update		first record	first record	
8	keyed_sequential_input		first record		
9	keyed_sequential_output				null
10	keyed_sequential_update		first record	first record	null
11	direct_input				
12	direct_output				null
13	direct_update			null	null

1. In the opening where no value is indicated for a designator, the designator is not relevant.

2. The use of certain options causes this to be initially set to beginning of file. See the description of the vfile_ I/O module in Subroutines manual for details.

TERMINAL INPUT/OUTPUT

Interactive terminals are normally connected to the system through the tty_ I/O module or the window_io_ I/O module.[1] The tty_ I/O module supports terminals in a typewriter-compatible manner. The window_io_ I/O module provides extended support for special video terminal features.

tty_ Support

The user's terminal is automatically "attached" to the tty_ I/O module during the course of process creation. Operations supported by the tty_ I/O module are described in the Subroutines manual.

window_io_ Support (the Video System)

The window_io_ I/O module is one of a number of software elements (I/O modules. commands, and subroutines) that compose the "video system." The two distinguishing capabilities of the video system are (1) its windowed display and (2) its real-time editor. The video software can be accessed from command level (via the window_call command) or via a subroutines interface provided by the subroutines window_ and video_utils_.

WHAT IS A WINDOW

A window is an area of the screen whose contents can be manipulated without affecting the rest of the display. For example, the user may scroll the contents of a segment in one window without moving the contents of the segment displayed on any other part of the screen.

Each window behaves like an individual video terminal. Many possible operations may be performed on a window. These include displaying characters, moving the cursor, erasing lines, inserting lines, and others. Characters are normally sent to a window via the Multics I/O system and the iox_ subroutine (see the Subroutines manual). Additional operations specific to the capabilities of video terminals, are performed by the window_ subroutine, which is analogous to iox_.

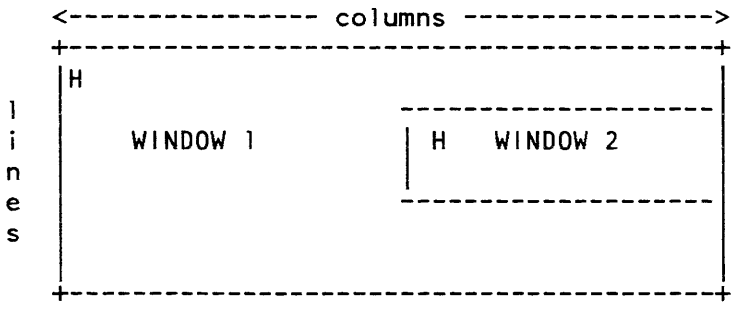
---

[1] Special_purpose I/O modules are provided for terminals connected to communications lines in which specialized protocols are in use. Such modules are described in the Subroutines manual.

The screen can be divided into several windows that can be viewed simultaneously but the windows may not overlap. The number of line and columns in each window can vary. A window can be one column wide or it can extend across the full width of the screen.

The size of a window is specified at the time the window is created. Character positions are identified by line and column with the origin (or home) located at the upper left hand corner of the window. Each window has its own home, line 1, column 1, and character positions are always with respect to the home of the specific window.

A screen divided into two windows is illustrated below.



*

WINDOW CAPABILITIES

The capabilities defined for a window are grouped into five categories: positioning the cursor, selective erasure, scrolling, selective alteration, and miscellaneous. Window operations may be performed with the window_call command or by a call to the window_ subroutine.

Positioning the Cursor

Each window has its own logical cursor. This cursor exists even when the terminal's cursor is performing operations in another window. The position of this cursor may be explicitly changed in a variety of ways. The cursor can be positioned absolutely or relatively. Absolute positioning can be to the home position or to an arbitrary line and column. Relative positioning can be up, down, left, or right any number of positions. The cursor also moves as characters are displayed in the window.

Selective Erasure

Selective Erasure (or clearing) means changing some region of the display so that no visible characters appear in that region, without changing any other area of the window. Most video terminals are capable of at least some selective erase operations. Where possible, the video system uses any special terminal features present to clear regions. When the terminal has no useful feature for clearing the specified region, regions are cleared by overwriting them with spaces. This can be a rather slow operation.

A region is a rectangle contained within a window. Like a window, it has an extent (height and width) and a position. All erasure operations pertain to regions. The definition of the region may be explicit (position and extent supplied in the call) or implicit (the region begins at the current cursor location, or at the home position). After the operation, the cursor is left in the upper left corner of the region.

A window may be cleared: entirely, from the home position to the end of the window; from the current cursor position to the end of the current line in the window; from the current cursor position to the end of the window. An arbitrary region may also be cleared.

Scrolling

A window may be scrolled up or down by a given number of lines. Scrolling up means moving lines up from the bottom of the window - deleting lines at the top, and adding new, blank lines at the bottom. Scrolling down means moving lines from the top of the window down, deleting at the bottom and adding at the top. Scrolling is usually done automatically by the video system when output fills the window, but it can also be requested explicitly.

Selective Alteration

Selective alteration means adding or deleting characters or lines in the middle of the window. When characters (or lines) are added, adjoining characters (or lines) move over to make room for the new ones. When characters (or lines) are deleted, characters (or lines) move in to fill up the gap. This differs from selective erasure, which only affects the characters erased.

Miscellaneous

Among other things, entries are provided in the window_ subroutine and the window_call command to sound an audible alarm, to obtain the current cursor position, and to output an arbitrary character sequence.

REAL-TIME EDITING

With real-time editing, all editing requests take effect immediately. The screen changes to show the effect of the characters or lines deleted. In addition, the set of editing characters expands to include several control characters.

Control characters are characters entered using the control key. The control key is a key that acts like the shift key. By itself it generates no characters; it is used to change the meaning of some other key. When the key "A" is typed while the control key is held down, the character sent by the terminal is control A, which is written as ^A. The control characters are the first 32 ASCII characters, 000 through 037 octal.

Alphabetic characters are given in capitals, but either an upper or lower case letter (as for N or n) can be used with default escape sequences. If an upper case letter is used with a user-defined sequence, both the upper and lower case keys must be bound in order for both keys to work. The letters ESC respresent the escape key. For ESC F, you would press the escape key, release it, and type an f or F.

Although most Multics users keep the system default erase (#) and kill (@) symbols, the video system recognizes and then assumes the values of any erase and kill characters that may have been set via the set_tty command.

The Erase Character

The erase character removes the character to the left of the cursor. The cursor moves to the left, and exactly one character is deleted. This is different from usual Multics editing where an erase character typed after white space deletes all whitespace, and otherwise deletes all characters from a column position. The erase character is settable for each window. In addition, the DEL character (\177) and the backspace character (\010) are always erase characters.

The Kill Character

The kill character deletes the entire line to the left of the cursor. The cursor then goes back to the beginning of the line. Again, this happens immediately. The deleted line is saved, and can be recovered. See "Retrieving Deleted Text" below. The kill character is settable per-window.

The Line Editor

Additional editing is possible using sequences of one and two characters. The two-character sequences all begin with the ASCII ESC character, (^], octal 033, \033), which is not the same as the Multics input escape character ("\").

Moving the Cursor

The line editor can move the cursor forward or backward within the current line while repositioning the cursor either a character at a time or a word at a time. A word is an unbroken string of uppercase and lowercase alphabetics, numerals, underscores, backspace characters, and hyphens. (This is the default definition of a word, which can be changed with the set_token_delimiters order, described in the window_io_ writeup.) The cursor can also move explicitly to the beginning or the end of the current line. The requests that perform these actions are listed under "Other Editor Requests" below.

Deleting Characters and Words

The line editor can delete a single character or an entire word at a time. Various editing requests described below can delete the character or word immediately to either the left or the right of the cursor. The deleted text (only words, not characters) is saved and can be retrieved. For example, typing ESC DEL (or ESC followed by the current erase character) deletes the word to the left of the cursor. The word is saved on the kill ring (see below).

Retrieving Deleted Text

Text deleted by the word and line kill characters is saved, and can be restored. The text is saved on a kill ring. A kill ring is a set of kill slots. Each slot holds deleted text. Successive word kills share one kill slot, so if several words are deleted one after another, all of them will be retrieved by a single retrieve command.

Deleted text is saved with previously deleted text if two kill requests are typed in succession. If intervening characters are typed, the kill ring is rotated: a new slot is selected to hold saved text.

Text is entered when the user types text followed by a carriage return. Each input line is added to the kill ring. This provides editing of the previous input line.

The following control characters are used to retrieve deleted text:

^Y
: (or yank) retrieves deleted text from the kill ring. This is the only way to recover from an erroneous kill character.

ESC Y
: can be typed only after either ^Y or ESC Y. It deletes the text just retrieved, without saving it on the kill ring, rotates the ring (to the next most recently killed text) and retrieves the text from the new top slot.

The following example is given in triplets. The first line shows what the user types, the second line shows what one line of the display looks like afterwards, and the third line (or lines) shows the kill ring. The top item on the kill ring is at the top of the column.

User Types:	This is a sentence
Display is:	This is a sentence
Kill Ring:	<empty>

NOTE: The kill ring is empty because the user has just invoked the video system.

User Types:	ESC DEL
Display is:	This is a
Kill Ring:	sentence

One word is deleted, and it begins the kill ring.

User Types:	ESC DEL
Display is:	This is
Kill Ring:	a sentence

Another word is deleted; it is merged into the same kill slot.

User Types:	an example sofa
Display is:	This is an example sofa
Kill Ring:	a sentence

User Types:	ESC DEL
Display is:	This is an example
Kill Ring:	sofa
	a sentence

This deleted word is not merged, because there has been typing since the last kill command. There are now two slots on the kill ring.

User Types:	of ^Y
Display is:	This is an example of sofa
Kill Ring:	sofa
	a sentence

The top kill slot is yanked back.

User Types:	ESC Y
Display is:	This is an example of a sentence
Kill Ring:	a sentence
	sofa

The kill ring is rotated, the previously yanked contents are deleted from the line, and the new top item from the ring is yanked to replace it.

If a carriage return were typed at the end of "This is an example of a sentence", the kill ring would then contain a new slot containing the entire input line.

Other Editor Requests

The following control characters are also recognized by the line editor. (Alphabetic characters are specified in upper case, but either upper case or lower case letters (e.g., ESC F or esc F can be used):

^L
: Clears the window and redisplays the input line.

^Q
: "quotes" the next character, causing it to have no special meaning. This is useful for entering control characters. It serves some of the same purposes as the input escape character (\).

^F
: moves the cursor forward one character.

^B
: moves the cursor backward one character.

ESC F
: moves the cursor forward one word.

ESC B
: moves the cursor backward one word.

ESC n control character
: repeat the specified action n times (e.g., ESC 6 ^D specifies that the next six characters are to be deleted).

^A
: moves the cursor to the beginning of the current line.

^E
: moves the cursor to the end of the current line.

^D
: deletes the current character (deletes forward).

DEL, #
: deletes the character to the left of the cursor (deletes backward).

ESC D
: deletes the current word (deletes forward).

ESC DEL, ESC #
deletes the word to the left of the cursor (deletes backward).

ESC C
capitalize initial word.

ESC U
capitalize word.

ESC L
lower case word.

ESC T
twiddle words. Transposes (interchanges) the last two words typed.

^T
twiddle characters. Transposes (interchanges) the last two characters typed.

^U
multiplies the next request four times (e.g., ^U^F moves forward four characters).

ESC ?
lists available window editor requests.

By default, no other control characters have meaning. If any are typed, the only action they cause is an audible alarm. You can create additional editor requests by writing PL/1 programs that conform to a standard calling sequence (see "Writing Editor Extensions").

The set of characters used to define a word for control characters such as ESC F can be changed via the set_token_characters control order. See the description in the window_io_ I/O module in the *Multics Subroutines* manual.

WRITING EDITOR EXTENSIONS

The video system provides a full input line editor, including the ability to edit in the middle of the line. Of course, there are many potential editor functions that people might like to use (see the *Emacs Text Editor User's Guide*), and not all of these are provided. Rather than attempt to anticipate every possible editor request, the video system allows users who are familiar with PL/1 to write their own editor requests and associate sequences of keystrokes (key bindings) with these requests.

The key binding mechanism can be used for a wide variety of applications. Since editor requests are executed immediately by single or multiple key stroke sequences, highly interactive facilities can be built into the input line editor.

Line Editor Routines

Editor request routines are PL/I programs that conform to a standard calling sequence. The request procedure is given complete control of the input buffer and can add or delete characters or modify the current contents of the buffer. The video system editor's redisplay facility manages all display updates; the individual editor routines need no knowledge of the video enviroment or the screen contents.

A library of editor utility routines is provided (see "Editor Utilities"). These can be called by user-written editor routines to perform such actions as insertion and deletion of text from the buffer, manipulation of the kill ring, and manipulation of words within the input buffer.

A line editor routine is declared as follows:

```
declare twiddle_words entry (pointer, fixed bin(35));

call twiddle_words (line_editor_info_ptr, code);
```

STRUCTURE ELEMENTS

line_editor_info_ptr
is a pointer to the line_editor_info data structure (described below).

code
is a standard status code. (Output) If the status code returned by the editor routine is error_table_$action_not_performed, the editor will ring the terminal bell to indicate that the editor routine was used improperly. Any other code will be reported in a more drastic manner, via the sub_err_ mechanism.

The line_editor_info structure (declared in window_line_editor.incl.pl1) is declared as follows:

```
dcl  1 line_editor_info          aligned based (line_editor_info_ptr),
       2 version                 char(8),
       2 iocb_ptr                pointer, /* to current window */
       2 repetition_count        fixed bin,
       2 flags,
         3 return_from_editor    bit(1) unaligned,
         3 merge_next_kill       bit(1) unaligned,
         3 old_merge_next_kill   bit(1) unaligned,
         3 last_kill_direction   bit(1) unaligned,
         3 numarg_given          bit(1) unaligned,
         3 suppress_redisplay    bit(1) unaligned,
         3 pad                   bit(30) unaligned,
       2 user_data_ptr           pointer, /* for user state info */
       2 cursor_index            fixed bin(21),
       2 line_length             fixed bin(21),
       2 input_buffer            character(1024) unaligned;
       2 key_sequence            character(128);

dcl line_editor_input_line char(line_editor_info.line_length)
     based (addr (line_editor_info.input_buffer));

dcl line_editor_info_version_2 char(8) static options (constant)
     init ("lei00002");
```

STRUCTURE ELEMENTS

version
: is the version string for this structure. (Input) The current version string, "lei00002", is the value of the variable line_editor_info_version_2, declared in the same include file.

iocb_ptr
: is the pointer to the current window. (Input)

repetition_count
: is the value of the numeric argument specified by the user, and is undefined if no numeric argument was specified (i.e., numarg_given flag = "0"b). (Input)

return_from_editor
: is a flag which is set by the editor routine if the editor invocation is to be terminated and the input line returned to the caller. The input buffer is redisplayed before the buffer is returned to the caller, unless overriden by the line_editor_info.suppress_redisplay flag. (Output)

merge_next_kill
is a flag which should be set when text is deleted and added to the kill ring if subsequent deletions are to be added to the same kill ring element. (Input/Output) This flag is managed by the editor utility routines. If the editor utility routines are used for all input buffer modifications, the user-written editor routine need never set this flag.

old_merge_next_kill (not used)
is an internal editor state flag and should not be modified.

last_kill_direction
is a flag indicating the direction of last kill. Off is forward; on is backward. This flag should not be modified.

numarg_given
returns "1"b (i.e. true if a numeric argument was supplied by the user via ESC-NNN or ^V).

suppress_redisplay
is a flag that stops the redisplay of the input buffer when line_editor_info.return_from_editor is set.

pad
reserved for future use.

user_data_ptr
points to a user data structure which the video system ignores, other than passing this pointer to requests that follow.

cursor_index
is the index of the character in the input buffer on which the cursor is currently located. (Input/Output) This index must be updated if characters are added or deleted before the cursor, or the cursor is moved by the editor routine. The cursor index must be no larger than one greater than the input_line_length. If the editor utility routines are used for all input buffer manipulations, the cursor_index will be updated appropriately.

line_length
is a count of the number of characters in the current input line. (Input/Output) This variable must be updated if any characters are inserted or deleted from the input buffer. The value of the line_length variable must always be non-negative, and must never be larger than the length of the input buffer. If the line editor utility routines are used for all input buffer manipulations, the line_length variable will be updated automatically.

input_buffer
is a character string containing the current input line. (Input/Output) Any manipulation may be performed on this string by the editor routine. It is recommended that the editor utility routines be used for all insertions and deletions to ensure that the various state variables and flags remain consistent. The line_editor_input_line variable can be used to address the valid part of the input buffer as a string.

key_sequence
a character string that contains the sequence of key strokes that invokes this editor routine.

Window Editor Utilities

As was mentioned above, a library of editor utility routines is provided for the benefit of user-written editor routines. Some operations can be performed simply by a user-written editor routine. For example, to position the cursor to the end of the line, simply set the cursor_index variable to one greater than the value of the line_length variable. However, most actions are more complex than this and it is recommended that the editor utility routines be used to perform most operations. The following is a description of these routines. In all cases, line_editor_info_ptr is the pointer to the editor data structure that is supplied as an argument to user-written editor routines.

```
dcl window_editor_utils_$insert_text entry (ptr, char(*), fixed bin(35));
call window_editor_utils_$insert_text (line_editor_info_ptr, "text",
     code);
```

Inserts the supplied character string into the input buffer at the current cursor location. If the string is too large to fit in the remaining buffer space, the code error_table_$action_not_performed is returned. This routine updates the line_length field of the line_editor_info structure, and the cursor_index if necessary.

```
dcl window_editor_utils_$delete_text entry (ptr, fixed bin,
     fixed bin(35));
call window_editor_utils_$delete_text (line_editor_info_ptr, count,
     code);
```

Deletes a specified number of characters (supplied by the variable count) from the input buffer at the current cursor location. If there are not enough characters remaining between the cursor and the end of the line, error_table_$action_not_performed is returned and no characters are deleted. The line_length component of the line_editor_info_structure is updated, and the cursor_index if necessary.

```
dcl window_editor_utils_$delete_text_save entry (ptr, fixed bin, bit(1),
     fixed bin(35));
call window_editor_utils_$delete_text_save (line_editor_info_ptr, count,
     kill_direction, code);
```

This entrypoint is identical to delete_text, but the deleted text is added to the kill ring. The kill_direction flag is used during kill merging to decide whether the killed text will be concatenated onto the beginning or end of the current kill ring element. "1"b is used to specify a forward kill (e.g. FORWARD_DELETE_WORD), "0" a backward kill.

```
dcl window_editor_utils_$move_forward entry (ptr, fixed bin,
     fixed bin(35));
call window_editor_utils_$move_forward (line_editor_info_ptr, count,
     code);
```

Advances the cursor forward a specified number of characters (supplied by the variable "count") in the input line. If there are not enough characters between the cursor and the end of the line, error_table_$action_not_performed is returned.

```
dcl window_editor_utils_$move_backward entry (ptr, fixed bin,
     fixed bin(35));
call window_editor_utils_$move_backward (line_editor_info_ptr, count,
     code);
```

Moves the cursor backward a specified number of characters (supplied by the variable "count") in the input line. If there are not enough characters between the cursor and the end of the line, error_table_$action_not_performed is returned.

```
dcl window_editor_utils_$move_forward_word entry (ptr, fixed bin(35));
call window_editor_utils_$move_forward_word (line_editor_info_ptr, code);
```

Updates the cursor_index to a position after the next word (or token) in the input line. A word is defined via the editor's set of token delimiters, set via the set_token_delimiters control order.

```
dcl window_editor_utils_$move_backward_word entry (ptr, fixed bin(35));
call window_editor_utils_$move_backward_word (line_editor_info_ptr,
     code);
```

Updates the cursor_index to a position before the preceeding word (or token) in the input line. A word is defined via the editor's set of token delimiters, set via the set_token_delimiters control order.

```
dcl window_editor_utils_$get_top_kill_ring_element  entry (ptr, char(*),
     fixed bin(35));
call window_editor_utils_$get_top_kill_ring_element (line_editor_info_ptr,
     text, code);
```

Returns the top kill ring element.

```
dcl window_editor_utils_$rotate_kill_ring entry (ptr, fixed bin(35));
call window_editor_utils_$rotate_kill_ring (line_editor_info_ptr, code);
```

Rotates the kill ring.

END-OF-WINDOW PROCESSING

When output has filled a window, old lines must be removed to make way for new ones. This is usually done by scrolling old lines off the top of the window. But for windows that cannot be scrolled (usually because the terminal cannot scroll) it is possible to move the cursor back to home, and output new lines overwriting the old ones. This is known as wrapped output. A variation on wrapped output is to clear the window after moving the cursor home. The action taken when a window is full is controlled on a per-window basis by any one or the following more_mode modes:

- clear
  the window is cleared, and output starts at the home position.

- fold
  output begins at the first line and moves down the screen a line at a time replacing exisitng text with new text. Prompts for a MORE response when it is about to overwrite the first line written since the last read or MORE break.

- scroll
  lines are scrolled off the top of the window, and new lines are printed in the space that is cleared at the bottom of the screen. This is the default for all terminals capable of scrolling (i.e., those terminals that have the capability to insert and delete lines).

- wrap
  output begins at the first line and moves down the screen a line at a time replacing existing text with new text. Prompts for a MORE response at the bottom of every window of output. This is the default for terminals incapable of scrolling.

MORE PROCESSING

As lines are displayed in the window, old lines are scrolled off the top of the window or otherwise removed. When output would cause a line to be removed that has been displayed since the most recent input, it is assumed that the user may not have had a chance to read it, and MORE processing occurs. The question "MORE? (RETURN for more; DEL to discard output)" appears on the screen, and no further output occurs until the user indicates that pending output is to be either displayed or discarded. MORE processing is controlled by the "more" mode, which is enabled by default.

Output resumes if the user strikes CR, and is discarded if the user strikes DEL. The characters used can be set by a control order. Type ahead characters are not seen by MORE processing. The response to MORE must be typed after the prompt appears. All other characters are buffered to be returned later.

When output is discarded, the video system simply ignores output until a get_line or get_chars call is made, a "reset_more" control order call is made, or the window is cleared, or the cursor is moved to home. WARNING: a prompt sent just before a get_line call will not be printed if output is discarded, unless the prompting program first issues a "reset_more" control order (or otherwise resets more processing).

OUTPUT BUFFERING

The video system sometimes buffers output internally, sending it to the terminal when certain internal conditions are satisfied. All buffered output is sent to the terminal whenever an input call is made (e.g., window_$get_echoed_chars). This ensures that all output, including prompts, is seen by the user before input is read. An application program that calls window_ entrypoints directly should take this buffering into account to perform correctly. If it is necessary to send output to the terminal when no read request is to be done (e.g., displaying an incremental message during a long computation), the application should call window_$sync on the I/O switch after the output has been requested (e.g., via a call to window_$overwrite_text). See the description of window_$sync in the window_ subroutine description in the Subroutines Manual.

STRUCTURE OF THE VIDEO SYSTEM

The video system is composed of various I/O modules, subroutines, and commands, as described below.

I/O Modules

The video system is divided into two layers, each implemented by an I/O module. The top layer, window_io_, makes terminal-independent calls to the lower level, tc_io_.

The window_io_ I/O module is responsible for translating window-relative calls (such as position to the beginning of the window) to terminal-relative calls (position to line 5, column 1, if that is where the window starts). The window_io_ I/O module is the video analogue of tty_. It supports control orders to change the size of its window, set the editing characters, read and set modes, etc. It also supports the basic iox_ operations of get_line, put_line, get_chars_, put_chars, etc.

The tc_io_ I/O module is responsible for all terminal-dependent support. It deals with padding, whitespace optimization, and optimal cursor movement. There is one instantiation of tc_io_ for every terminal under the control of the video system. Although implemented as an I/O module, the only operations tc_io_ supports are attach, detach, and control; it is intended as an internal interface for use by window_io_ only.

Subroutines

The video_utils_ subroutine controls invocation and revocation of the video system in a process. It revokes the terminal's attachment to tty_ and attaches the user_I/O switch to a window that covers the entire screen. Upon revoking the video system, it reattaches the terminal to tty_.

The window_ subroutine is the main user interface to the video system. Its entry points define operations which a video terminal might reasonably be expected to have, such as "position cursor" or "delete lines." By calling these entries and letting the video system take care of determining which sequence of characters will effect the desired operation, the applications writer can write programs which will function identically on a wide range of terminals without having to worry about what those terminals are.

The COBOL and FORTRAN programmer can utilize specialized subroutines (cb_window_ and ft_window_) to obtain video management capabilities.

Command

The window_call command is the command-level interface to the video system.

USING THE VIDEO SYSTEM

The following subsections described basic video operations as implemented from the command-level interface (window_call) and the subroutine interface (window_). (The examples of command usage are embedded in an exec_com).

Attaching the Video System

The video system must be checked to determine if it is turned "on." It is not likely that novice users would do this initially but it might be included in a project start_up. If it is on, it is important that to leave it alone. If it is turned on again an error message is produced. If the video system is turned on, then the application should use the space allocated to the user_input/output window instead of the whole screen. Thus, if the user creates a separate window for interactive messages, an application should not use that space. Using the space allocated to the user_io window respects the user's explicit wishes and prevents violation of the restriction against using two overlapping windows at the same time.

When the video system is invoked, the entire screen is covered by a window associated with the user_i/o I/O switch. The user must determine how much of the screen he has and divide up that amount for use by the application. Since terminals vary in the length of the screen, and some users already may have some lines devoted to their own video display, there are probably less than 20 available lines, so design with that in mind. As long as there are eight or ten lines available for user input/output that should be sufficient.

The first step then is for the user to determine whether or not the video system is turned on and, if not, turn it on. This should be included at the beginning of all applications. The following is the exec_com example. The lines are numbered only for the purpose of explanation and the numbers should not be included in the exec_com.

```
1    &set already_video &[window_call video_invoked]
2    &if &[not &(already_video)]
3    &then window_call invoke
4    &set first_line &[window_call get_first_line]
     &set n_lines &[window_call get_window_height]
```

where:

1. determines whether or not the video system is attached to the user's terminal.
2. turns it on if it isn't already on.
3. invokes window_call initiating the window environment.
4. sets the lines for the window. This is part of the first step because when you revoke the video system at the end of the exec_com, you must set the screen to the size it was originally.

The following is the PL/I example that does the same thing. Again, the lines are numbered for the purpose of explanation and the numbers should not be included in the program.

```
dcl   (addr, null) builtin;

dcl   iox_$control entry (ptr, char (*), ptr, fixed bin (35));
dcl   com_err_ entry () options (variable);
dcl   iox_$user_io ptr ext static;
dcl   video_utils_$turn_on_login_channel entry
        (fixed bin (35), char (*));
dcl   video_data_$terminal_iocb ext static ptr;

dcl   ME char (32) init ("test_program") static options (constant);
dcl   code fixed bin (35);
dcl   already_video bit (1);
dcl   reason char (128);

1     %include window_control_info;
2     dcl 1 my_window_info like window_position_info;
3     my_window_info.version = window_position_info_version_1;
4     if video_data_$terminal_iocb = null () then do;
5        call video_utils_$turn_on_login_channel (code, reason);
6        if code ^= 0 then do;
              call com_err_ (code, ME, "^a", reason);
              return;
         end;
7        already_video = "0"b;
      end;
8     else already_video = "1"b;
9     call iox_$control (iox_$user_io, "get_window_info",
         addr (my_window_info), code);
10    if code ^= 0 then do;
          call com_err_ (code, ME, "get_window_info.");
          return;
       end;
```

where:

1. includes appropriate structure declarations
2. declares an automatic copy of window info
3. sets the version number of window info
4. determines if the video system is not activated then does 4 through 6
5. turns on the video system and
6. if there is an error, reports it to the caller and quits
7. makes a note to the effect that video was invoked by this program
8. goes to here if the video system is already activated (video was not activated by this program)
9. gets the current size and location (beginning line number) of the user_i/o window
10. prints error message

Detaching the Video System

At the end of the session, the video system can be detached. First, is the exec_com example for revoking the video system. The lines are numbered only for the purpose of explanation and these numbers should not be included in the exec_com.

```
1     &if &(already_video)
2     &then window_call change_window -line &(first_line)
        -height &(n_lines)
3     &else window_call revoke
```

where:

1. determines whether or not video was activated by this exec_com.
2. if video was activated by another exec_com, then user_i/o window is returned to previous size and it is cleared.
3. otherwise, the window interface to the video system is deactivated and the user_i/o window goes to full screen.

Next is the PL/1 example:

```
1     if already_video  then do;
2     call video_utils_$turn_off_login_channel(code);
        if code ^=0 then do;
            .
            .
            .
        end;
      end;
3     else do; call iox_$control(iox_$user_io, "set_window_info",
        addr(my_window_info), code);
        if code ^=0 then do;
            .
            .
            .
        end;
      end;
```

where:

1. determines whether or not video was activated by this program.
2. if the video system was activated by this program, it is then deactivated and the user_i/o window goes to full screen.
3. if video was previously attached, then the user_i/o window is returned to its previous size.

Design Requirements for Windows

As part of the menu design process, the user must decide ahead of time how the display will look and from that determine the number of windows that will be advantageous.

As an example, the screen could be divided into three windows. The top window could display the status of the user with the user name, a description of what the user is doing and a clock. The middle window could contain various menus and could grow or shrink depending on the selection made. The bottom window could be for unformatted output and for typing in input.

The number of windows technically permitted is quite large and probably more than one will need. Knowing how many functions are to be performed, the user should carefully select the number of windows to be used by an application. It is possible on a 24 line terminal to have 24 windows but rarely, if ever, would that be useful. Each window would be too small and the screen would be too cluttered. Practically, there should not be more than five. Windows should not overlap. Each window has its own extent (height and width) and location (the position of its home on the screen). Windows can change their extent and location as long as they never overlap. The initial extent and location of a window is determined in the attach description of the window.

Create Window Operation

The creation and definition of windows is done with arguments to window_call or with the entry points of the window_ subroutine. The first action discussed is create_window. Part of the creation process is the naming of windows. Windows are associated with iox_ I/O switches. The "name of the window" is just the name of the switch, or as it is sometimes called, the iocb name. Since many Multics commands and subroutines make use of the standard switches user_io, user_input, error_output, and user_output, it is usually necessary to have these switches connected to some window. This is done by window_call invoke or video_utils_$turn_on_login_channel. By convention, the bottom window of the screen is used for user_i/o.

Important Window Requests

Before a window can be created the user must decide on its starting line number as discussed above in "Attaching Video" and its length (in number of lines). As mentioned earlier, it is customary to get space for a new window from the user_i/o window and to position the new window at the top of the user_i/o window. Therefore, one of the first things to do is find out where the user_i/o window is. Once this is known determine just how high, in lines, the new window must be and shrink the user_i/o window by that amount. It is a good idea to always check to make sure there is enough space left in the user_i/o window to allow meaningful communication once it has been shrunk. In our examples we will insist on at least a five line user_i/o window.

To do all that has been discussed so far in an exec_com, we would have the following:

```
&- stored in the default value segment as the_menu.

&set io_start &[window_call get_first_line]
&set io_height &[window_call get_window_height]
&set menu_height &[menu_describe the_menu -height]

&- Now calculate the new positions of both windows.

&set menu_start &(io_start)
&set io_start &[plus &(io_start) &(menu_height)]
&set io_height &[minus &(io_height) &(menu_height)]

&- The label referenced below would, of course, need to be
&- defined and would include an appropriate error message.

&if &[nless &(io_height) 5]
  &then &goto USER_I/O_TOO_SMALL

&- Now shrink user_i/o

window_call change_window -line &(io_start) -height &(io_height)

&- And define the new window, called able

window_call create_window -io_switch able -line &(menu_start) -height
  &(menu_height)
```

The real work of creating the new window above was done by the window_call command with the create_window argument. This command created the necessary iox_ I/O switch attachments to make "able" an I/O switch which describes a video system window that occupies the first "menu_height" lines of what was user_i/o.

To do the same thing in PL/I you would use the following code fragment:

```
/* Get the variables initialized.  We assume that the menu has */
/* been created and that the variable called the_menu points   */
/* to the menu.                                                */

%include menu_dcls;
%include window_control_info;

dcl 1 menu_needs like menu_requirements;
dcl 1 menu_window_info like window_position_info;
dcl 1 io_window_info like window_position_info;
```

```
/* Get information about size of the menu. */

menu_needs.version = menu_requirements_version_1;
call menu_$describe (the_menu, addr(menu_needs), code);
  if code ^= 0 then do;
       .
  process the error
       .
  end;

/* Get information about size of the user_i/o window. */

io_window_info.version = window_position_info_version_1;
call iox_$control (iox_$user_io, "get_window_info",
  addr(io_window_info), code);
  if code ^= 0 then do;
       .
  process the error
       .
  end;
menu_window_info = io_window_info;

/* Now calculate the new positions of both windows. */

menu_window_info.origin.line = io_window_info.origin.line;
menu_window_info.extent.height = menu_needs.lines_needed;
io_window_info.origin.line = io_window_info.origin.line +
  menu_needs.lines_needed;
io_window_info.extent.height = io_window_info.extent.height -
  menu_needs.lines_needed;
if io_window_info.extent.height < 5 then do;
       .
  complain that user_i/o window
  is too small
       .
  end;

/* Now shrink user_i/o window. */

call iox_$control (iox_$user_io, "set_window_info",
  addr(io_window_info), code);
  if  code ^= 0 then do;
       .
  process the error
       .
  end;
```

```
/* Create an I/O switch by which the menu window will be */
/* referenced.                                            */

menu_io_switch_name = "menu_i/o." || unique_chars_ ("0"b);
call iox_$find_iocb (menu_io_switch_name, menu_window_iocbp, code);
  if code ^= 0 then do;
      .
  process the error
      .
  end;

/* And define the new menu window */

call window_$create (video_data_$terminal_iocb,
  addr(menu_window_info), menu_window_iocbp, code);
  if code ^= 0 then do;
      .
  process the error
      .
  end;
```

Change Window Operation

In the above examples it was necessary to change or shrink the user_i/o window in order to create a new window. When we discuss destroying windows below we will see a need to expand the user_i/o window to recover the space freed by the destruction of a window.

Command level changes are done with the window_call keyword change_window. In PL/I the changes are made by the set_window_info control order. In general this will be preceded by a get_window_info control order and some calculations.

Destroy Window Operation

Once a window is no longer needed it must be destroyed, i.e., the I/O switch must be closed and detached thus freeing up the space on the screen that was occupied by the window. In addition, this space should be returned to some active window so that it can be used. If the freed space is adjacent to the user_i/o window it should be consumed by that window, but it can be added to any adjacent window. In our examples we will add it back to user_i/o.

To reverse the effects of the exec_com window creation example above we would have:

```
&- destroy the able window

window_call delete_window -io_switch able

&- and let user_i/o have the space back

&set io_start &(menu_start)
&set io_height &[plus &(menu_height) &(io_height)]
&set menu_start 0 menu_height 0

window_call change_window -line &(io_start) -height &(io_height)
```

In PL/I we would have:

```
/* destroy the able window */

call window_$destroy (...);
   if code ^= 0 then do;
        .
   process the error
        .
   end;

/* and let user_i/o have the space back */

io_window_info.origin.line = menu_window_info.origin.line;
io_window_info.extent.height = menu_window_info.extent.height
     +io_window_info.extent.height;

call iox_$control (iox_$user_io, "set_window_info",
     addr (io_window_info), code);
   if code ^= 0 then do;
        .
   process the error
        .
   end;
```

Clear Window Operation

Another very useful operation is the clear_window operation. This clears the entire window to all spaces and leaves the cursor positioned at the upper left hand corner of the window. There are other clearing operations, but this one is the simplest and most useful.

From command level we can clear the user_i/o window by:

window_call clear_window

If we had wanted to clear, say the able window, we would have included the -io_switch control argument specifying able as the window to operate on.

This same effect, clearing the able window of our examples, can be accomplished from PL/I by:

```
call window_$clear_window (menu_window_iocbp, code);
   if code ^= 0 then do;
        .
   process the error
        .
   end;
```

The clear_window operation is useful when an application wants to start with a clean slate in the user_i/o window. For example, before printing out a description of some menu option it might be desirable to clear the user_i/o window.

MAGNETIC TAPE INPUT/OUTPUT

Magnetic tape input/output operations in Multics Release 11.0 differ from those of previous releases. The two methods of performing tape input/output are described below.

Magnetic Tape Input/Output in Releases Previous to MR 11.0

Prior to MR 11.0, tape I/O was performed through iox_ calls to one of four I/O modules, as follows:

```
tape_ansi_
tape_ibm_
tape_mult_
tape_nstd_
```

The individual I/O modules each supported operations specific to a particular tape format. The user had to attach the appropriate I/O module depending on the file format in which the tape was to be read or written.

The tape I/O modules as well as the iox_ subroutine are described in the Subroutines manual.

Magnetic Tape Input/Output in MR 11.0

In MR 11.0, the mtape_ I/O module provides an alternative method for processing tapes in ANSI or IBM format. (The tape_ansi_ and tape_ibm_ I/O modules remain available for use.)

The mtape_ I/O module is called via an attach description made through the iox_ subroutine. The mtape_ I/O module supports three new iox_ entries: iox_$open_file, iox_$close_file, and iox_$detach.

The new entrypoints allow control parameters to be passed to the open, close, and detach entries. The open_file entry in particular allows the tape file name, file position and file attributes to be given when the file is opened instead of with the attach description (as is done in the tape_ansi_ and tape_ibm_ I/O modules). This allows several tape files to be processed during a single attachment. The mtape_ I/O module as well as the iox_ subroutine is described in the Subroutines manual.

The mtape_ I/O module currently supports two tape formatting standards, IBM and ANSI. Format-specific processing operations are performed by externally-callable subroutines known as per-format modules. Current per-format modules are: ansi_tape_io_ and ibm_tape_io_. Selection of the appropriate per-format module is performed by mtape_ based on information returned by RCP after a successful volume mount, and on the presence of the -volume_type attach description argument for mtape_. The per-format modules are described in the Subroutines manual.

All control arguments that are part of the mtape_ attach and detach descriptions and all control arguments that are specific to each per-format module's open and close description are supplied with default values. The user can find the system-supplied default value for any of the above entities by referring to the appropriate description in the Subroutines manual. The default control arguments, as well as other argument processing information, are stored in the data space of a standard value segment. A user can set (as well as display and delete) the default control arguments by means of user commands. See the descriptions of the mtape_set_defaults, mtape_get_defaults and mtape_delete_defaults commands in the Commands manual.

BULK INPUT AND OUTPUT

The Multics system has provisions for three types of bulk I/O: high-speed printer output, punched card input, and punched card output.

Printed Output

The enter_output_request command causes the contents of a Multics file (segment or multisegment file) containing Multics ASCII characters to be printed on a

high-speed printer. See the description of the enter_output_request command in the Commands manual. See also the dprint command in Commands manual.

The printed output has the following parts:

1. Header sheet. This sheet identifies: the requesting access_id; the person and destination of the person to whom the dprint is sent; the pathname of the file; the date, time, and day of the week the file was printed; the physical device on which the file was printed; and the installation identifier. If more than one copy of the file is requested, the number of the copy (in the form "copy m of n" where m and n are numbers from 1 to 30 for eor or 4 for dprint) is indicated on the header sheet. Each corner of the header sheet contains the sequence number of the printed output. If more than one copy of the file is requested, the header sheet of each duplicate copy has the same sequence number.

2. Announcement page. This page may be used by the installation to send a message to all users. The dprint is folded so the header sheet is always an outside page and the announcement page is an inside page. Except for duplicate copies of the same segment, the header sheet and announcement page are separated by four lines of overstruck characters printed on the paper perforation; these separator lines and the sequence number of the printed output assist in filing output.

3. File contents. The contents of the file are printed in a format determined by the characteristics of the physical device or by control arguments to the enter_output_request (or dprint) command, and also by escape sequences in the text if escape processing is being performed. See the enter_output_request (or dprint) command in the Commands manual for explicit details on formatting output.

4. Summary sheet. This sheet indicates: the date, time, and day the output was requested; the date, time, and day the output was printed; the request type; the queue; the physical device; the number of lines and pages in the printed output; and the cost per 1000 lines and 1000 pages; the total cost of the output and the access_id to which it is charged. The summary sheet also identifies the pathname of the file, the entryname of the file, and the destination to which the output is sent. The sequence number of the printed output is also in each corner of the summary sheet. The printed output is folded so the summary sheet is always an outside page.

VERTICAL FORMAT CONTROL

The printer software supports an escape (esc) mode that allows users to control the vertical format of their data by inserting printer control escapes in the text. The escape sequence can be used to make the printer slew to a specified line on each sheet of paper, similar to a slew to VFU pattern. The mechanisms for specifying an escape sequence, specifying channel stops and causing escape processing to take place are the same for remote terminal printers as for local site printers.

There are 16 logical channels that can be set for each line of the paper. Physical page lengths up to 127 lines are supported. The administrator uses the request type info segment to set "esc" mode and to define which of the 16 channels are associated with each line number. There may be one request type info segment for each request type of generic type "printer." The contents of the request type info segment for a given request type may be printed on the terminal by the command:

```
display_prt_rqti >ddd>idd>rqt_info_segs>SEG
```

where SEG is the entryname of the request type info segment. For more information on the display_prt_rqti command, see the *Multics Bulk I/O Manual,* Order No. CC34. (By convention, the entryname is <request_type>_info, e.g., printer_info for request type printer.)

Within the request type info segment the channel definitions might be described as follows:

```
Line  (1):     1, 5, 15;
Line (11):     5;
Line (21):     5;
Line (30):    15;
    .           .
    .           .
    .           .
```

This means that a slew to channel 1 moves the paper to line 1 (This is the same as a new page or form feed character). Similarly, the escape sequence <esc>c5<etx> (slew to channel 5) moves the paper to line 1, a second slew to channel 5 moves to line 11, then to line 21. In general, if the printer is on line X when it receives a slew to channel <n>, it searches forward starting at line X+1 until it finds a line with channel <n> set and moves the paper to that line. If a channel stop for channel <n> is not defined, the printer advances to the next line.

The move to channel <n> escape control sequences are specified in the text of a print file by:

```
<esc>c<n><etx>
```

where:

typ <esc>
: is the ASCII ESC character (octal 033).

c
: is the lowercase ASCII character "c" which defines this to be a channel slew control sequence.

<n>
: is an integer that defines the target channel of the slew control sequence.
($1 \leq n \leq 16$)

<etx>
is the ASCII ETX character (octal 003).

If esc mode has been set in the rqt_info_seg, this control causes the printer to move to the next line associated with channel <n>.

Punched-Card Output

The enter_output_request command described in the Commands manual causes the contents of Multics files to be punched. See also the dpunch command in the Commands manual. The files can be punched under mcc, raw, or 7punch conversion modes. See Appendix C for more information on punched card output. Files punched by a remote station will be punched under rmcc mode.

Punched-Card Input

Facilities are provided to read punched card decks into Multics files. There are several conventions for interpreting the punched codes used in a user's card deck. The central site reader is capable of reading any punch codes, including binary data. Remote terminal card readers normally cannot read binary code. There are four types of card formats that can be input to Multics: Multics card codes (mcc), remote Multics card code (rmcc), 7punch, and raw.

mcc
: The Multics card code is defined in "Punched Card Codes" in Appendix C of this document. It consists of a superset of the EBCDIC card punch codes and can be produced by 029 key punches. Each column is interpreted as one character. The 12-bit card codes are converted to 9-bit ASCII codes.

rmcc
: Remote Multics card code does not concern itself with punch codes, but rather with the characters that are transmitted. Selection of punch code is determined on the basis of hardware configuration. Conversion and translation is specified by the -terminal_type control argument to the remote_input_ I/O module. For more information see the description of the remote_input_ I/O module in the Subroutines manual. Punch codes are not specified, unlike mcc format, because various remote terminals use different codes for the same characters, and it is the character, not the punch code, that is transmitted.

7punch
: The 7punch decks are binary representations of existing files with checksums and sequence numbers added, and the data portions of the cards are read in exactly as they were punched out. The format of a 7punch deck is described in Appendix C.

raw

Raw decks are simply read into Multics files without any conversion, and without regard to format; that is, the 960 bits on each card are read into the file in column order and without padding. Any desired conversion can then be performed by the user.

The flip cards prepared when a deck is punched (described in Appendix C) and other sorts of labeling cards from other systems are not read correctly and should be removed from decks. See Appendix C for more information on punched card input.

There are two modes of Multics punched card input: bulk data input and remote job entry (RJE). Bulk data input is used to copy data from punched cards into the Multics storage system. Remote job entry on Multics is a mechanism that allows a registered user to submit an absentee job from a card deck.

ACCESS REQUIRED FOR CARD INPUT

In order for a user to submit a card deck for input to Multics, the following conditions must be met:

1. The user must be registered for card input and have an assigned card input password set up by the system administrator, or have been given permission to use the null password feature.

2. A special access control segment must exist in the user's mailbox directory. Proper access must be set for the station in order for it to read card decks (see "Card Input Access Control Segment" later in this section.)

3. The user must have permission to use the card input station. This is granted by the system administrator on the ACL of the station access control segment.

For RJE jobs, the tag portion of the process group ID of the absentee process (which is used in access control calculations) is "p". A system administrator or a user may deny access to RJE jobs with the ACL term:

```
null *.*.p
```

or similar ACL terms, assuming that there does not exist a more specific ACL term that gives access.

Card Input Registration and Password

Each user usually must be given a card input password by the system administrator in order to use any form of card input on Multics. The card input password defined should be different from the user's interactive password. The Person_id and password of the user are provided on control cards at the time the deck is submitted.

The user who submits card input must include a password card as the second card of his deck. It has the form:

```
++PASSWORD xxxxxxxx
```

where the xxxxxxxx is the user's registered card input password (1-8 characters). It is customary to turn the keypunch printer off when punching the password. Users who have r access to `>scl>rcp>card_input_password.acs` do not need to be registered for card input. In this case, xxxxxxxx should be blanks.

If the Person_id given in the ++DATA or ++RJE card is not registered appropriately, or if the password given on the password card is incorrect, the input is not accepted. If the password is not specified and the mode is bulk data input, and the user has r access to `>scl>rcp>card_input_password.acs`, the input is allowed. In this case the person need not even be registered. The submitter must have access to the card input and station access control segments defined below.

Card Input Access Control Segment

The card input access control segment allows a user to control which stations can be used to read bulk card input using his Person_id and Project_id. The pathname of this access control segment is:

```
>udd>Project_id>Person_id>card_input.acs
```

This segment must exist with an ACL containing read access to each station that is permitted to submit bulk data input for the user and execute access for each station that is permitted to submit RJE jobs. For example:

```
re Station *.*
```

The ACL star convention may be used in the normal fashion. If the user job lacks access to the card input ACS, input is not accepted. If this segment does not exist or if the access is not as specified, card input will not be permitted. In addition, the user must have permission to use the station, with the same type of access as defined above, granted by the system administrator on the ACL of the station access control segment as discussed below. Remote terminal login is accepted only from remote terminals that have a registered station ID and password. The name of each registered station and its password is stored in the person name table.

Station Access Control Segment

Each station has an access control segment in the directory `>system_control_1>rcp>named station.acs`. The ACL of this segment lists all users allowed to submit card input through the station; a user must have read access for bulk data input and execute access for RJE. For example:

```
re Person_id.Project_id.*
```

The ACL star convention may be used in the normal fashion. If the user job lacks access to the station ACS, the input is not accepted.

This check allows a site to specify that a certain station is reserved for the use of a certain group of users. The ACS can also be used to ensure that certain stations are not used to submit card input for privileged users, such as *.SysAdmin, who should never use the facility for reasons of security. If a user is not on the ACS for a station he wishes to use, he should contact the System Adminstrator to obtain proper access.

CONTROL CARD INFORMATION

Control cards are used to tell the card input process how to read the user's data and what to do with it. Each control card consists of a key (e.g., ++FORMAT) and possibly some data fields. The control card key must start in column 1 and may not contain any spaces. Data fields are separated from the key and from each other by one or more spaces.

All letters punched on the control cards are mapped to lowercase except those immediately following an escape character (backslash or cent sign). For example, `\SMITH.\SYS\MAINT` is mapped into Smith.SysMaint.

At the central site, users submit card decks to operations personnel for processing. At remote sites that have a card reader terminal, the user may have to physically place the card deck in the reader. In the latter case, the user must be sure to include some additional control cards that must be placed before and after the user's card deck. For more information on these additional control cards (++EOF and ++UID, for example) see the *Multics Bulk Input/Output*, Manual Order No. CC34.

BULK DATA INPUT

Bulk data input is the mode of card input used to read a punched card deck and write its contents into a card image segment in the Multics storage system. The user is able to read the card image segment from his normal Multics process (interactive or absentee.) For security reasons, card image segments are created in system pool storage rather than in the user's directory. Once the data has been read, the user may copy the card image segment into his directory using the copy_cards command (see the description of this command in the Commands manual).

Card image segments must be copied from the system pool storage within a reasonable time, as these segments are periodically deleted from the system pool.

The user identified on the ++DATA card is notified by mail when his card deck has been successfully read.

A complete card deck for bulk data input is shown below.

Control Card Format of a Card Deck for Bulk Data Input

```
++DATA DECK_NAME  PERSON_ID  PROJECT_ID
++PASSWORD  PASSWORD
++CONTROL  OVERWRITE
++AIM  ACCESS CLASS OF DATA CARDS
++FORMAT  PUNCH_FORMAT  MODES
++INPUT
    .
    .
(user data cards)
```

The only control cards required are the first which is an identifier card, the second which is a password card and the last which is the end of control input. For an explanation of all the control cards refer to Appendix C of this manual.

The user should submit a complete card deck to operations. The deck must follow the format specified in the card input section in Appendix C.

Normally, the access_class is system_low and the ++AIM cards can be omitted. However, if the access_class is greater than system_low, the ++AIM cards are required.

REMOTE JOB ENTRY

Remote job entry is used to copy standard Multics commands punched on cards into an absentee input segment and have it submitted as an absentee job (as though it were done from an interactive process). The user's card deck is copied into an absentee input segment created in the normal system pool storage used for bulk data input card image segments. When the user's deck has been successfully read, an absentee request is submitted on behalf of the user who provided the deck. A special header is added to the absentee input segment so that a dprint request of the absentee output segment is automatically generated using the request type associated with the remote terminal or the request type of the local printer, depending on the input device.

Format of a Card Deck for Remote Job Entry

```
++RJE  DECK_NAME  PERSON_ID  PROJECT_ID
++PASSWORD  PASSWORD
++AIM  ACCESS CLASS OF ABSENTEE PROCESS
++RJECONTROL  CONTROL ARGS TO THE EAR COMMAND
++RJEARGS  ARGUMENTS FOR THE ABSENTEE PROCESS
++EPILOGUE  COMMAND
++FORMAT PUNCH_FORMAT MODES
++INPUT
   .
   .
(user absentee file)
```

The only cards required are the first which is an identifier card, the second which is a password card and the last which is the end of control input. For an explanation of all the control cards refer to Appendix C of this manual.

The user should submit a complete card deck to operations. The deck must follow the format specified in the card input section in Appendix C.

Normally, the access_class is system_low and the ++AIM cards can be omitted. However, if the access_class is greater than system_low, the ++AIM cards are required.

REMOTE JOB ENTRY WITH FOREIGN COMPUTER SYSTEMS

Multics provides facilities for users to submit card decks to a remote computer system for execution and to receive output from that execution either for printing/punching locally or for online perusal. This section describes the mechanisms available for using this facility.

Submitting Card Decks to a Remote System

Each card deck to be transmitted to a remote system for execution must be contained in a separate Multics segment. This segment can be created using an editor, bulk card input, or any other appropriate mechanism.

The segment must consist of ASCII text only; no binary data (object segments, and so on) can be included. The exact format of the segment's contents is dependent on the remote system being accessed and should be determined from the appropriate documentation for the remote system.

To transmit the segment to the remote system, issue the enter_output_request or dpunch command, specifying the mcc conversion mode and the request type established by your system administrator explicitly for this purpose. A separate request type will be used for each remote system to which card decks can be submitted.

For example, to submit the card deck contained in the segment "sample.cdc" in the working directory to a remote CDC system, deleting the deck after it is successfully transmitted, issue the command:

```
enter_output_request -mcc -rqt cdc_jobs -dl sample.cdc
```

where "cdc_jobs" is the request type established by your system administrator to submit decks to the CDC system.

Receiving Output from a Remote System

By default, printed and punched output returned by a remote system to Multics is automatically printed or punched locally. However, your system administrator may decide that the returned output should be made available to users for online inspection.

If output is to be available for online inquiry, each output file must contain Multics control records that establish the identity of the user who owns the file. Either the job control language (JCL) submitted to the remote system or the programs(s) executed on the remote system must be modified to cause the required control records to appear in the output files. Check with your system administrator to determine which mechanism must be used for each remote system.

Returned output files that are to be available for online examination are placed in system pool storage where they may be retrieved using the copy_cards command described in the Commands manual. Output files must be copied in a reasonable time, as they are periodically deleted from the system pool.

Format of an Output File Transmitted to Multics for Online Perusal

```
++IDENT FILE_NAME PERSON_ID PROJECT_ID
++FORMAT MODES
++CONTROL OVERWRITE AUTO_QUEUE
++INPUT
   .
   .
(output data)
   .
   .
<EOF record>
```

The only user-supplied control records required are ++IDENT and ++INPUT. For an explanation of these control records, refer to Appendix C of this manual.

Each output file is delimited by an end-of-file (EOF) record supplied automatically by the remote system. All control records in the output file from ++IDENT through ++INPUT inclusive and the EOF record are removed from the file before it is placed into pool storage.

For printed output, each paper motion command in the file is translated into the character sequence that will best simulate the requested motion when (and if) the file is printed locally via the enter_output_request or dprint command. The exact character sequences used are given in Table 5-6.

One of the paper motion commands that may be received is a request to skip to a specific printer channel stop. This command is converted to a logical channel slew sequence as defined in "Vertical Format Control" earlier in this section. The user should check the RQTI segment of the request type used for printing the output file to determine which channel stops may be used in the output file. (The program executed on the remote system is responsible for placing this particular paper motion command in the output file. The exact mechanism used to do this should be determined from the appropriate documentation for the remote system.)

Table 5-6. Translations of Paper Motion Commands in Output Files

Paper motion command	Character sequence
Slew zero lines	CR (octal 015) (1)
Slew one line	NL (octal 012)
Skip to channel N	ESC c <N> ETX (2)

1. Overprint the current line with the previous line.

2. This sequence is octal 033, octal 143, the decimal representation of the channel number encoded as ASCII characters (e.g., octal 061, octal 065 for channel #15) and octal 003.

IMPLEMENTATION OF INPUT/OUTPUT MODULES

The information provided below is applicable to writing I/O modules. It describes the format and function of I/O control blocks, and provides a list of implementation rules. For descriptions of the iox_ entry points, refer to the Subroutines manual.

Some instances in which a user might wish to create a new I/O module are given below:

1. Pseudo Device or File. An I/O module could be used to simulate I/O to/from a device or file. For example, it might provide a sequence of random numbers in response to an input request. The discard_ system I/O module (described in the Subroutines manual) is an example of this sort of module.

2. New File Type. An I/O module could be used to support a new type of file in the storage system, such as a file in which records are accessed via a hash table. Another example is an I/O module which supports Honeywell GCOS standard tape formats.

3. Reinterpreting a File. An I/O module could be designed to overlay a new structure (relative to the standard file types) on a standard type of file. For example, an unstructured file might be interpreted as a sequential file by considering 80 characters as a record. The record_stream_ I/O module does this type of interpretation.

4. Monitoring a Switch. An I/O module could be designed to pass operations along to another module while monitoring them in some way (e.g., by copying input data to a file). The audit_ system I/O module (described in the Subroutines manual) is an example of this sort of I/O module.

5. Unusual Devices. Working through the tty_ I/O module (described in the Subroutines manual) in the raw mode, another I/O module might transmit data to/from a device that is not a standard Multics device type (as regards character codes, etc.).

The last three items listed illustrate a common arrangement. The user attaches an I/O switch, x, using an I/O module, A. To implement the attachment, module A attaches another switch, y, using another I/O module, B. When the user calls module A through the switch x, module A in turn calls module B through the switch y. Most nonsystem I/O modules that perform true I/O work in this way, because a nonsystem I/O module (or some module that it calls) in turn calls a system I/O module. There are system I/O routines at a more primitive level than the I/O modules, but user-written I/O modules ordinarily do not call these routines.

I/O Control Blocks

An I/O switch is embodied by an I/O control block (IOCB) that is created when iox_$find_iocb is called the first time the I/O switch is referenced by name. The control block remains in existence for the life of the process unless explicitly destroyed by a call to iox_$destroy_iocb.

The principal components of an I/O control block are pointer variables and entry variables whose values describe the attachment and opening of the I/O switch. There is one entry variable for each I/O operation with the exception of the attach operation, which does not have an entry variable since there can be only one attach entry point in an I/O module. To perform an I/O operation through the switch, the corresponding entry value in the control block is called. For example, if iocb_ptr is a pointer to an I/O control block, the call:

```
call iox_$put_chars (iocb_ptr, buff_ptr, buff_len, code);
```

can be thought of as:

```
call iocb_ptr->iocb.put_chars (iocb_ptr, buff_ptr, buff_len, code);
```

All routines must call the iox_ subroutine, as the internal representation of the control block may change.

I/O CONTROL BLOCK STRUCTURE

The declaration given below describes the first part of an I/O control block. Only those few I/O system programs that use the remainder of the I/O control block declare the entire block. Thus, all references to I/O control blocks here refer only to the first part of the control block. For example, the statement "no other changes are made to the control block" means that no other changes are made to the first part of the control block, and so on. The I/O system might make changes to the remainder of the block, but these are of interest only to the I/O system. For full details on the entry variables, see the descriptions of the corresponding entries in the iox_ subroutine in the Subroutines manual and the iox_$init_standard_iocbs entry point in this manual. This structure is declared in the iocb.incl.pl1 include file.

```
dcl  1 iocb                  aligned based,
       2 version             character (4) aligned,
       2 name                char (32),
       2 actual_iocb_ptr     ptr,
       2 attach_descrip_ptr  ptr,
       2 attach_data_ptr     ptr,
       2 open_descrip_ptr    ptr,
       2 open_data_ptr       ptr,
       2 reserved            bit (72),
       2 detach_iocb         entry (ptr, fixed (35)),
       2 open                entry (ptr, fixed, bit (1) aligned,
                                    fixed (35)),
       2 close               entry (ptr, fixed (35)),
       2 get_line            entry (ptr, ptr, fixed (21), fixed (21),
                                    fixed (35))
       2 get_chars           entry (ptr, ptr, fixed (21), fixed (21),
                                    fixed (35)),
       2 put_chars           entry (ptr, ptr, fixed (21), fixed (35)),
       2 modes               entry (ptr, char (*), char (*), fixed (35)),
       2 position            entry (ptr, fixed, fixed (21), fixed (35)),
       2 control             entry (ptr, char (*), ptr, fixed (35)),
       2 read_record         entry (ptr, ptr, fixed (21), fixed (21),
                                    fixed (35)),
       2 write_record        entry (ptr, ptr, fixed (21), fixed (35)),
       2 rewrite_record      entry (ptr, ptr, fixed (21), fixed (35)),
       2 delete_record       entry (ptr, fixed (35)),
       2 seek_key            entry (ptr, char (256) varying, fixed (21),
                                    fixed (35)),
       2 read_key            entry (ptr, char (256) varying, fixed (21),
                                    fixed (35)),
       2 read_length         entry (ptr, fixed (21), fixed (35)),
       2 open_file           entry (ptr, fixed bin, char (*),
                                    bit (1) aligned, fixed bin (35)),
       2 close_file          entry (ptr, char (*), fixed bin (35)),
       2 detach              entry (ptr, char (*), fixed bin (35));

declare iox_$iocb_version_sentinel character (4) aligned external static;
```

ATTACH POINTERS

If the I/O switch is detached, the value of iocb.attach_descrip_ptr is null. If the I/O switch is attached, the value is a pointer to the following structure:

```
dcl 1 attach_descrip based  aligned,
    2 length                fixed bin(17),
    2 string                char (0 refer (attach_descrip.length));
```

The value of attach_descrip.string is the attach description. See the attach description earlier in this section.

If the I/O switch is detached, the value of iocb.attach_data_ptr is null. If the I/O switch is attached, the value may be null, or it may be a pointer to data used by the I/O module that attached the switch. To determine whether the I/O switch is attached or not, the value of iocb.attach_descrip_ptr should be examined; if it is null, the switch is detached.

OPEN POINTERS

If the I/O switch is closed (whether attached or detached), the value of iocb.open_descrip_ptr is null. If the switch is open, the value is a pointer to the following structure:

```
dcl 1 open_descrip based    aligned,
    2 length                fixed bin(17),
    2 string                char (0 refer (open_descrip.length));
```

The value of open_descrip.string is the open description. It has the following form:

```
mode  {info}
```

where:

1. mode
   is one of the opening modes (e.g., stream_input) listed below. The modes and their corresponding numbers are:

```
 1   stream_input
 2   stream_output
 3   stream_input_output
 4   sequential_input
 5   sequential_output
 6   sequential_input_output
 7   sequential_update
 8   keyed_sequential_input
 9   keyed_sequential_output
10   keyed_sequential_update
11   direct_input
12   direct_output
13   direct_update
```

   The include file iox_modes.incl.pl1 declares a set of named constants for these mode values.

2. info
   is other information about the opening, such as the open description. If info occurs in the string, it is preceded by one blank character.

If the I/O switch is closed, the value of iocb.open_data_ptr is null. If the I/O switch is open, the value may be null, or it may be a pointer to data used by the I/O module that opened the switch. The iox_modes.incl.pl1 include file gives standard names and named constants for the opening modes.

ENTRY VARIABLES

The value of each entry variable in an I/O control block is an entry point in an external procedure. When the I/O switch is in a state that supports a particular operation, the value of the corresponding entry variable is an entry point that performs the operation. When the I/O switch is in a state that does not support the operation, the value of the entry variable is an entry point that returns an appropriate error code. The iox_ subroutine provides four error entries that set the error code argument for the I/O module entry to an appropriate error_table_ value. The entries and the corresponding error codes are:

```
iox_$err_not_attached          (error_table_$not_attached)

iox_$err_not_closed            (error_table_$not_closed)

iox_$err_no_operation          (error_table_$no_operation)

iox_$err_not_open              (error_table_$not_open)
```

SYNONYMS

When an I/O switch named x is attached as a synonym for an I/O switch named y, the values of all entry variables in the I/O control block for x are identical to those in the I/O control block for y with the exception of iocb.detach. Thus a call:

```
call iox_$<operation> (x_iocb_ptr,...);
```

immediately goes to the correct routine.

The values of iocb.open_descrip_ptr and iocb.open_data_ptr for x are also the same as those for y. Thus, the I/O routine has access to its open data (if any) through the I/O control block pointed to by x_iocb_ptr.

The value of iocb.actual_iocb_ptr for x is a pointer to the control block for the last switch in a chain of switches that have been connected to each other by the syn_ I/O module. (When the switch x is not attached as synonym, this pointer points to the control block for x itself.) I/O modules use this pointer to access the actual I/O control block whose contents are to be changed, for example, when a switch is opened. The I/O system then propagates the changes to any other control blocks that have been attached as synonyms to the actual I/O control block.

Writing an I/O Module

The information presented in the following paragraphs pertains to the design and programming of an I/O module. In particular, conventions are given that must be followed if the I/O module is to interface properly with the I/O system. The reader should be familiar with the material presented under the headings "Multics Input/Output System" and "File Input/Output," in this section as well as with the iox_ subroutines description in the Subroutines manual.

DESIGN CONSIDERATIONS

Before programming begins on an I/O module, the functions it is to perform should be clearly specified. In particular, the designer should list the opening modes to be supported and consider the meaning of each I/O operation supported for those modes. See also Table 5-1, which describes what operations should be supported for each opening mode.) (See "Open Pointers" above for a list of opening modes.) The specifications in the description of the iox_ subroutine must be related to the particular I/O module (e.g., what seek_key means for the discard_ I/O module).

The designer should decide what the attach, open, close and detach descriptions will be. The attach description defines all the information needed to attach an I/O switch through the I/O module to a specific device and/or volume. The open description gives all information relating to a specific file (a set of data on the volume), its file location and attributes. The close description gives the disposition information for the file or position information for the volume after the file is closed. The detach description gives the disposition of the device or volume after the I/O switch is detached. If open, close and detach descriptions are not needed, then the designer should implement a nonextended I/O module, because it is easier to interface nonextended I/O modules with language I/O facilities. If open, close and detach descriptions are needed, an extended I/O module must be implemented to accept these descriptions.

An I/O module contains routines to perform attach, open, close, and detach operations and the operations supported by the opening modes. Typically, though not necessarily, all routines are in one object segment. If the module is in a bound segment, only the attach entry need be retained as an external entry. Other routines are accessed through entry variables in I/O control blocks.

An I/O module may have several routines that perform the same function but in different situations (e.g., one get_line routine for stream_input openings, another for stream_input_output openings). Whenever the situation changes (e.g., at opening), the module stores the appropriate entry values in the I/O control block.

IMPLEMENTATION RULES

The following rules apply to the implementation of all I/O operations. Additional rules that are specific to a particular operation are given later. In the rules, iocb is a based variable declared as described under "I/O Control Blocks" above, and iocb_ptr is an argument of the operation in question.

1. For most operations, the usage (entry declaration and parameters) of a routine that implements an I/O operation is the same as the usage of the corresponding entry in the iox_ subroutine. See the Subroutines manual for details on the iox_ subroutine. Exceptions are the attach, open_file, close_file, and detach operations. The I/O module calling sequences for these operations are described below.

2. Except for attach and detach, the actual I/O control block to which an operation applies (i.e., the control block attached by the called I/O module) must be referenced using the value of `iocb_ptr->iocb.actual_iocb_ptr`. It is incorrect to use just iocb_ptr, and it is incorrect to remember the location of the control block from a previous call (e.g., by storing it in a data structure pointed to by iocb.open_data_ptr).

3. On entry to an I/O module, the value of iocb_ptr->iocb.open_data_ptr always equals the value of:

   ```
   iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr
   ```

   The value of iocb_ptr->iocb.open_descrip_ptr always equals the value of:

   ```
   iocb_ptr->iocb.actual_iocb_ptr->iocb.open_descrip_ptr
   ```

   Thus, the data structures related to an opening may be accessed without going through iocb.actual_iocb_ptr. However, if you need to free or reallocate the structures pointed to by iocb.open_data_ptr and iocb.open_descrip_ptr, you must reset these variables using `iocb_ptr->iocb.actual_iocb_ptr` as shown above, and you must follow the procedures in step 4 below to propagate these changes to synonym switches.

4. If an I/O operation changes any values in an I/O control block, the changes must be made in the actual I/O control block (Rule 1 above). I/O modules should mask IPS signals when the iocb is being modified, to prevent asynchronous event call handlers from using an iocb which is in an inconsistent state. To mask IPS signals:

   a. Get ready to change the iocb by copying all pointers or entry constants that the new iocb will contain into automatic variables. This will snap links to lessen the probability of a linkage error while interrupts are masked.

b. Establish an any_other handler to abort the operation (by closing the iocb if you were opening it when a failure occurs, or by detaching the iocb if you were attaching it) and unmask IPS signals.

c. Execute the call:

```
call hcs_$set_ips_mask (0, mask);
```

The routine hcs_$set_ips_mask is used to disable one or more IPS interrupts. (See the description of hcs_$set_ips_mask in the Subroutines manual.)

d. Change the iocb by setting the iocb pointers and entry variables appropriate to the operation being performed, as described below.

e. Execute the call:

```
call iox_$propagate (iocb_ptr);
```

where iocb_ptr points to the changed control block. The routine iox_$propagate reflects changes to other control blocks attached as synonyms. It also makes certain adjustments to the entry variables in the control block when the I/O switch is attached, opened, closed, or detached.

f. Execute the call:

```
hcs_$reset_ips_mask (mask, mask);
```

This routine is used to enable one or more IPS interrupts. (See the description of hcs_$reset_ips_mask in the Subroutines manual.)

g. Revert the any_other handler.

5. The procedure entrypoints supporting all I/O operations must be external entry points. Only the attach entrypoint must be retained in a bound segment containing the I/O module.

ATTACH OPERATION

The name of the routine that performs the attach operation is derived by concatenating the word "attach" to the name of the I/O module (e.g., discard_attach is the name of the attach routine for the discard_ I/O module). Each attach routine has the following usage:

```
declare module_nameattach entry (ptr, (*)char(*) varying, bit(1)
     aligned, fixed bin(35));

call module_nameattach (iocb_ptr, option_array, com_err_switch,
     code);
```

where:

1. iocb_ptr
   points to the control block of the I/O switch to be attached. (Input)

2. option_array
   contains the options in the attach description. (Input) If there are no options, its bounds are (1:0). You should check for no options using the PL/I statement:

   ```
   if dimension (option_array,1) = 0 then ...
   ```

   Otherwise, its bounds are (1:N) where N is the number of options. iox_ divides the attach description string into the individual options which are passed in this option_array.

3. com_err_switch
   indicates whether the attach routine should call the com_err_ subroutine (described in the Subroutines manual) when an error is detected. (Input)

   ```
   "1"b    yes
   "0"b    no
   ```

4. code
   is a standard status code. (Output) The code indicates the success or failure of attachment, with any nonzero value indicating failure.

The following rules apply to coding an attach routine:

1. If the I/O switch is already attached (i.e., if `iocb_ptr->iocb.attach_descrip_ptr` is not null), return the code error_table_$not_detached; do not make the attachment.

2. If, for any reason, the switch is not and cannot be attached, return an appropriate nonzero code and do not modify the control block. Call the com_err_ subroutine if, and only if, com_err_switch is "1"b. Optionally, sub_err_ may be called if com_err_switch is "0"b, and an error code is not a sufficient description of the problem. A sub_err_ flag of ACTION_DEFAULT_RESTART should be used. If sub_err_ returns, then the I/O module should return the error code to the caller. If the attachment can be made, follow the remaining rules and return with code set to 0.

3. For a nonextended I/O module (which does not support open, close and detach descriptions), set `iocb_ptr->iocb.open` and `iocb_ptr->iocb.detach_iocb` to the appropriate open and detach_iocb routines. For an extended I/O module, set `iocb_ptr->iocb.open_file` and `iocb_ptr->iocb.detach` to the appropriate open_file and detach routines. Be sure to follow step 4 of the "Implementation Rules" above when changing the I/O control block.

4. Set `iocb_ptr->iocb.attach_descrip_ptr` to point to a structure as described in "I/O Control Blocks" above. The attach description in this

structure must be fabricated from the options in the argument option_array. Options may be modified in the attach description (e.g., a pathname option may be expanded), and default values should be included in the attach description for omitted options.

5. If desired, set `iocb_ptr->iocb.attach_data_ptr`, `iocb_ptr->iocb.modes`, and `iocb_ptr->iocb.control`. Make no other modifications to the control block.

6. Call iox_$propagate.

OPEN OPERATION

An open operation is performed only when the actual I/O switch is attached but not open. The open routine for a nonextended I/O module has the same calling sequence as the iox_$open subroutine. The open_file routine for an extended I/O module has the following calling sequence:

```
declare open_file_routine entry (ptr, fixed bin, (*) char(*)
     varying, bit(1) aligned, fixed bin(35));

call open_file_routine (iocb_ptr, mode, option_array, unused,
     code);
```

STRUCTURE ELEMENTS

iocb_ptr
: points to the control block of the I/O switch to be opened. (Input)

mode
: is the number assigned to the opening mode to be used for this opening. (Input) See iox_modes.incl.pl1 for a list of acceptable values and for named constants associated with the values.

option_array
: contains the options in the open description. (Input) If there are no options, option_array bounds are (1:0). You should check for no options using the PL/I statement:

```
if dimension (option_array,1) = 0 then ...
```

Otherwise, its bounds are (1:N) where N is the number of options. iox_ divides the open description string into the individual options which are passed in this option_array.

unused
: must be "0"b. (Input)

code
: is a standard status code. (Output) The code indicates the success or failure of opening, with any nonzero value indicating failure.

The following rules apply to coding both open and open_file routines:

1. If, for any reason, the opening cannot be performed, return an appropriate code and do not modify the I/O control block. For example, if the switch is not attached (i.e., `iocb_ptr->iocb.attach_descrip_ptr=null`), then return a code of error_table_$not_attached. If the switch is already open (i.e., `iocb_ptr->iocb.open_descrip_ptr^=null`), then return a code of error_table_$not_closed. If an incorrect or unsupported opening mode is given, return a code of error_table_$bad_mode. If an error occurs in an open description option, return an error code appropriate to the error. Optionally, sub_err_ may be called to report the error if a simple error code isn't sufficient to describe the error. A sub_err_ flag of ACTION_DEFAULT_RESTART should be used. If sub_err_ returns, then the I/O module should return the error code to the caller. If the opening can be performed, follow the remaining rules and return with code set to 0.

2. Set `iocb_ptr->iocb.actual_iocb_ptr->iocb.op` to an appropriate routine. This applies for each operation allowed for the specified opening mode. Be sure to follow step 4 of the "Implementation Rules" above when changing the I/O control block. The following is a list of possible I/O operations:

```
close or close_file
get_line
get_chars
put_chars
read_record
write_record
rewrite_record
delete_record
seek_key
read_key
read_length
modes
position
control
```

   iocb.close must be set for nonextended I/O modules, while iocb.close_file must be set for extended I/O modules. Refer to Table 5-1 for a list of operations which should be allowed for each possible opening mode.

3. If either the modes operation or the control operation is enabled with the I/O switch attached but not enabled when the switch is open, set `iocb_ptr->iocb.actual_iocb_ptr->iocb.op` (where op is modes or control) to iox_$err_no_operation.

4. Set iocb.open_descrip_ptr to point to a structure as described in "I/O Control Blocks" above.

5. If desired, set `iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr`. Do not make any other modifications to the control block.

6. Call iox_$propagate.

CLOSE OPERATION

A close operation is performed only when the actual I/O switch is open. The close routine for a nonextended I/O module has the same calling sequence as the iox_$close subroutine. The close_file routine for an extended I/O module has the following calling sequence:

```
declare close_file_routine entry (ptr, (*) char(*) varying, fixed
     bin(35));

call close_file_routine (iocb_ptr, option_array, code);
```

STRUCTURE ELEMENTS

iocb_ptr
points to the control block of the I/O switch to be closed. (Input)

option_array
contains the options in the close description. (Input) If there are no options, option_array bounds are (1:0). You should check for no options using the PL/I statement:

```
if dimension (option_array,1) = 0 then ...
```

Otherwise, its bounds are (1:N) where N is the number of options. iox_ divides the close description string into the individual options which are passed in this option_array.

code
is a standard status code. (Output) The code indicates the success or failure of closing with any nonzero value indicating failure.

The following rules apply to coding both close and close_file routines:

1. The close routine should set the bit counts on modified segments of a file, free any storage allocated for buffers, etc., and in general, clean things up. It should then free any per-opening data structures, and set `iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr` to null, if this pointer is nonnull. Be sure to follow step 4 of the "Implementation Rules" above when changing the I/O control block.

2. For a nonextended I/O module, set the following to the appropriate open and detach_iocb routines:

```
iocb_ptr->iocb.actual_iocb_ptr->iocb.open
iocb_ptr->iocb.actual_iocb_ptr->iocb.detach_iocb
```

For an extended I/O module, set the following to the appropriate open_file and detach routines:

```
iocb_ptr->iocb.actual_iocb_ptr->iocb.open_file
iocb_ptr->iocb.actual_iocb_ptr->iocb.detach
```

Set iocb_ptr->iocb.actual_iocb_ptr->iocb.open_descrip_ptr to null.

3. If either the modes operation or the control operation was enabled with the switch open and should be not enabled with the switch closed, set `iocb_ptr->iocb.actual_iocb_ptr>iocb.op` (where op is modes or control) to iox_$err_no_operation. If the operation was not enabled with the switch open but should be enabled with the switch closed, set the entry variable to the apppropriate routine.

4. Do not make any other modifications to the control block.

5. Call iox_$propagate.

6. The close routine must not return without closing the switch.

DETACH OPERATION

A detach operation is performed only when the actual I/O switch is attached but not open. The detach_iocb routine for a nonextended I/O module has the same calling sequence as the iox_$detach_iocb subroutine. The detach routine for an extended I/O module has the following calling sequence:

```
declare detach_routine entry (ptr, (*) char(*) varying,
     fixed bin(35));

call detach_routine (iocb_ptr, option_array, code);
```

STRUCTURE ELEMENTS

iocb_ptr
points to the control block of the I/O switch to be detached. (Input)

option_array
contains the options in the detach description. (Input) If there are no options, option_array bounds are (1:0). You should check for no options using the PL/I statement:

```
if dimension (option_array,1) = 0 then ...
```

Otherwise, its bounds are (1:N) where N is the number of options. iox_ divides the detach description string into the individual options which are passed in this option_array.

code
is a standard status code. (Output) The code indicates the success or failure of detaching with any nonzero value indicating failure.

The following rules apply to coding both detach_iocb and detach routines:

1. Set `iocb_ptr->iocb.attach_descrip_ptr` to null. Free any per-attachment data structure, and set `iocb_ptr->iocb.attach_data_ptr` to null. Be sure to follow step 4 of the "Implementation Rules" above when changing the I/O control block.

2. Do not make any other modifications to the control block.

3. Call iox_$propagate.

4. The detach routine must not return without detaching the switch.

MODES AND CONTROL OPERATIONS

These operations can be accepted with the I/O switch attached but closed; however, modes or control operations are usually meaningful only when the switch is open. When this is the case, modes and control operations should be allowed only when the switch is open.

If the control operation is supported, it must return the code error_table_$no_operation when given an invalid order. In this situation, the state of the I/O switch must not be changed.

If the modes operation is supported, it must return the code error_table_$bad_mode when given an invalid mode. In this situation, the state of the I/O switch must not be changed.

PERFORMING CONTROL OPERATIONS FROM COMMAND LEVEL

Most of the operations supported by an I/O module may be used directly from command level by using the io_call command (see the Commands manual). When a control operation requires an info structure (see the description of the iox_$control entry points in the Subroutines manual), a special interface, the "io_call" order, is used to make these control operations from command level possible. All standard I/O modules that implement control operations requiring info structures should implement this interface, as described below.

When an io_call command of the form:

```
io_call control switch_name {optional_args}
```

is issued, the io_call command performs an "io_call" control operation to the switch specified using the following info structure (found in io_call_info.incl.pl1):

```
dcl 1 io_call_info              aligned based (io_call_infop),
      2 version                 fixed bin,
      2 caller_name             char(32),
      2 order_name              char(32),
      2 report                  entry options (variable),
      2 error                   entry options (variable),
      2 af_returnp              ptr,
      2 af_returnl              fixed bin (21),
      2 fill (5)                bit(36),
      2 nargs                   fixed bin,
      2 max_arglen              fixed bin (21),
      2 args                    (0 refer  (io_call_info.nargs))
                                 char (0 refer
                                (io_call_info.max_arglen))
                                 varying;
```

STRUCTURE ELEMENTS

version
is the version number of this structure, currently 1.

caller_name
is the name of the caller (normally io_call) to be used in any error

order_name
is the order specified in the command line.

report
is an entry like ioa_ to be called to report the results of the order.

error
is an entry like com_err_ to be called to report any errors.

af_returnp
is a pointer to the active function return string if the io_call command was invoked as an active function.

af_returnl
is the maximum length of the active function return string.

nargs
is the number of optional_args specified in the command line.

max_arglen
is the length of the longest argument.

args
is an array of the actual arguments from the command line.

The I/O module, upon receipt of an io_call order, should do the following:

1. If io_call_info.order_name specifies an order that requires an info structure with input values, the I/O module should use io_call_info.args to determine what data should be placed into the info structure. Once the structure is complete, the I/O module should call iox_$control, passing it io_call_info.order_name and a pointer to the info structure just created. Exactly how io_call_info.args is to be interpreted in order to build the info structure depends on the I/O module and what order is being performed. This should be documented along with the I/O module.

2. If io_call_info.order_name specifies an order that requires an info structure with output values, the I/O module should call iox_$control passing it io_call_info.order_name and a pointer to a structure of the appropriate kind. Then, using io_call_info.report, the I/O module should display the results of the control operation in some meaningful way. It is possible in this case that io_call_info.args could be used for control arguments to determine exactly what will be displayed. As in input type orders, the interpretation of these arguments is completely at the discretion of the I/O module.

3. If io_call_info.order_name specifies an order that does not require an info structure, or is an invalid order, then the I/O module should return error_table_$undefined_order_request. The io_call command, seeing this code, will call iox_$control again, this time passing the original control order name, and a null info_ptr.

4. If the I/O module detects an error in handling an io_call order, it must do one of two things. First, it may return an error code, in which case io_call prints an error message. Secondly, it may call io_call_info.error (used like the com_err_ subroutine) to report the error directly. In this case, a zero error code should be returned to the caller. The latter choice is recommended, especially in cases where the I/O module can print a more informative error message.

I/O modules that do not support control operations that require info structures need not implement the io_call order at all. The io_call order can be rejected along with all other invalid orders in which case the order is performed with a null info_ptr by the io_call command as described in item 3 above.

Control operations can also be performed through the active function interface of the io_call command. In this case, the mechanism is basically the same with the following differences:

1. The order issued by the io_call command is io_call_af, not io_call.

2. Instead of printing a result, the I/O module should store its result in the varying string defined by io_call_info.af_returnp and io_call_info.af_returnl.

The io_call_af order should only be supported for orders that have meaning as an active function. As in the io_call order, the interpretation of io_call_info.args is completely up to the I/O module.

OTHER OPERATIONS

Routines for the other operations are called only when the actual I/O switch is attached and open in a mode for which the operation is allowed, the opening and attachment having been made by the I/O module containing the routine. The following modifications to the I/O control block of the actual I/O switch can be made:

1. Reset `iocb_ptr->iocb.actual_iocb_ptr->iocb.open_data_ptr`.

2. Reset an entry variable set by the open routine, e.g., to switch from one put_chars routine to another.

3. Close the switch in an unrecoverable error situation. In this case, the rules above for the close operation must be followed.

If any change is made to the iocb, be sure to follow step 4 of the "Implementation Rules" above.

OUTER MODULES

The iox_ I/O module with which user_i/o is attached at process initialization is called the outer module. In order to support reconnection of terminals, I/O modules used as outer modules must respect certain conventions. For an example of the appropriate techniques, examine the source of tty_.

All outer modules must support the -login_channel attach control argument, to mean that the switch will be connected to the device specified by user_info_$terminal_data.

When the user is disconnected, the special condition sus_ is signalled in the process. The program sus_signal_handler_ catches the condition, and blocks awaiting notification from the Answering Service that a new terminal is available. This may happen at any time, even when the process is compute-bound. When sus_signal_handler_ receives the notification, it searches the attach table for all switches with the control argument -login_channel in their attach description. Each one is closed, detached, attached, and opened.

The result of this is that an outer module may be interrupted in the middle of an operation, have its switch detached and closed, and be left to continue execution. Outer modules must be designed to avoid failure under these circumstances. An outer module may mask the sus_ IPS signal for the duration of all operations affecting the attachment data structures, but there is only a limited amount of CPU time available after the signal. If sus_signal_handler_ does not make the proper response to the Answering Service within this time, the process is terminated.

The alternative strategy is to detect asynchronous detachments. This can be done using a half lock in the attach data. As any operation is started, the half lock has one added to its value. When an operation is completed, one is subtracted. If the detach or close entrypoints are called and find a nonzero half lock, they may not free any storage that may be referenced by interrupted operations. Instead, they set flags in the attach data indicating that an asynchronous close or detach has taken place. When any of the other entrypoints detect these bits, they assume that a new attachment has been made, and call iox_ on the new attachment to complete their operation. Then they return.

For example, if tty_'s put_chars operation gets an error indicating that the process no longer has permission to use the terminal, it checks for the asynchronous bits. If they are not present, it blocks to await the arrival of the sus_ signal. If they are, it calls iox_$put_chars on its actual_iocb, and returns the results it returns.

RESOURCE CONTROL PACKAGE

The Resource Control Package (RCP) provides a mechanism for device reservation, assignment, and attachment.

Relationship of RCP to Other I/O Facilities

Input/output in the user environment of the Multics system is organized around the user-ring I/O system subroutine, iox_. The entry points of iox_ provide for a general, device-independent interface supporting I/O and control functions. They may be called either via explicit PL/I code or via the facilities of language-provided I/O. Often, they are called internally from programs that deal with peripheral I/O.

The user-ring I/O system is organized around I/O modules, programs that support the iox_ interfaces for a specific device, class of devices, or class of operations upon a given device or class of devices. (The available interfaces of iox_ are described in the Subroutines manual.) I/O modules make appropriate calls upon the I/O interfaces of the supervisor, the resource control package (RCP), and the I/O interfacer to arrange for use of peripheral devices and perform operations upon them. The system provides a repertoire of I/O modules for peripheral devices. These I/O modules are documented in the Subroutines manual. The user may provide his own I/O modules as well (see above).

RCP is responsible for allocation and deallocation of peripheral devices to user processes. By means of RCP, user processes (and I/O modules) can gain access to peripheral devices. RCP provides for access checking and device selection. RCP is described in detail below.

The I/O interfacer (IOI) is the facility of the supervisor through which user programs (via I/O modules) can operate peripheral devices. IOI provides for the operation of the I/O hardware and the multiplexing of channels and other physical resources between processes. IOI can only be used to manipulate a device once a process has acquired the right to use that device via RCP.

The user can construct device-specific DCW lists and call IOI to initiate the I/O operation. When the operation completes, IOI provides the user with a wakeup and the status. The hardware protection and relocation features of the IOM are used by IOI to allow the user complete control over his DCW lists and data with no possibility of damaging the system.

The interrelationship between user code, iox_, RCP, IOI, and the I/O modules is shown in Figure 5-1.

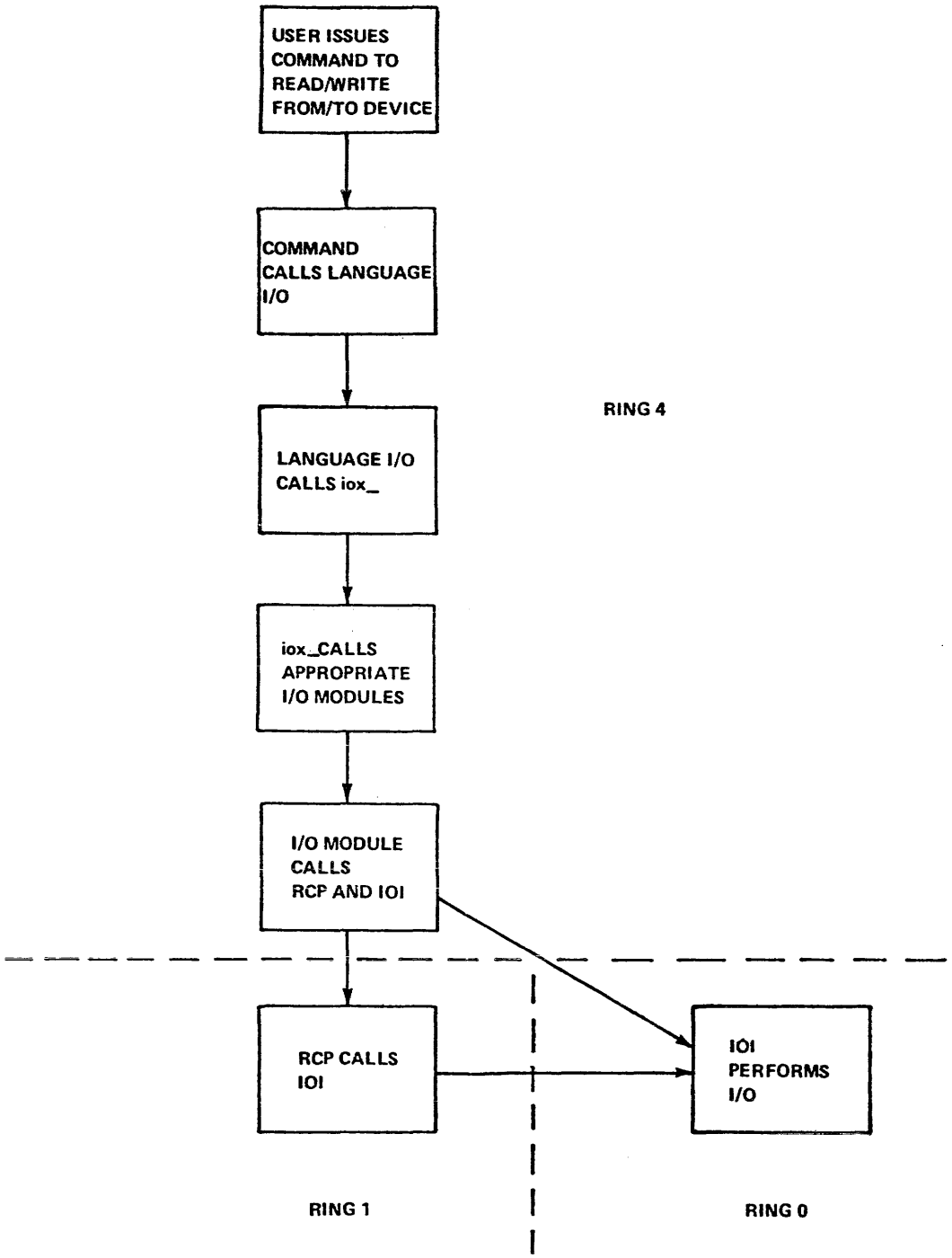


Figure 5-1. Interrelationship between User Code, iox_, RCP, IOI, and the I/O Module

Summary of RCP Actions

The resource control functions performed by RCP are:

1. reservation/cancellation
2. assignment/unassignment
3. attachment/detachment

These functions are summarized below.

RESERVATION, ASSIGNMENT, AND ATTACHMENT

The functions reserve, assign, and attach are organized into hierarchical levels. Defaults are provided at each level so that users not desiring to exercise features specific to a level do not have to concern themselves with that level.

```
1  reserve
   2  assign
      3  attach
      3  detach
   2  unassign
1  cancel
```

The first level involves the reservation of resources by processes. Tape drives, disk drives, tape volumes and disk volumes can be reserved. Reservations are process-specific and remain in effect until the process requests a cancellation. Reservation implies that a process temporarily has exclusive rights to a resource. This exclusive right means that no other process can use that resource for the duration of the reservation. Reservation does not necessarily imply that a resource is actually being used. Multiple resources can be reserved with one reservation.

Assignment, like reservation, is process-specific and lasts until unassignment or process termination. Any resource type can be assigned. An assignment also gives a process temporary exclusive rights to a device. Assignment does not necessarily mean that a device is currently being used. That is the function of the next level, attachment. Only one resource can be assigned per assignment.

A resource cannot be used until it is attached. When RCP is called to attach a resource, it initiates communication with the ring 0 subsystem that actually provides the use of the resource. Before the attachment is completed, RCP performs all initialization necessary to allow the attaching process to begin using the resource. For devices, this involves attaching the device via IOI and making sure that the device is ready and that any volume needed has been determined to be accessible, mounted, and authenticated.

The hierarchical relationship among reservation, assignment, and attachment implies that a higher-level function (e.g., reservation) can stand alone, while a lower-level function (e.g., attachment) can only be performed after all higher-level functions have been performed. RCP can perform the following device reservation, assignment, and attachment functions:

1. Reserving a resource. This means that no other process can use it during this period of time.

2. Explicitly assigning a reserved device. The device is assigned to a process but is not attached.

3. Attaching an explicitly assigned device.

4. Attaching an unassigned device. Since a device cannot be attached until it is assigned, RCP automatically reserves and assigns the device and then performs the attachment. The device is said to be implicitly assigned.

5. Detaching an implicitly assigned device. After the device is detached, RCP automatically unassigns the device.

6. Detaching an explicitly assigned device. The device is detached but is not unassigned.

7. Explicitly unassigning a device. If the device is attached, it is first detached and then unassigned.

8. Cancelling reservation of a resource.

The rules stated above imply that I/O modules do not have to be concerned with the assignment or unassignment of devices. They need to be concerned with only the attachment and detachment of a device. RCP, however, does allow the above rules to be overridden. When detaching a device an I/O module can tell RCP to retain the device assignment regardless of whether the device was explicitly or implicitly assigned.

When a process terminates, RCP automatically detaches and unassigns all devices currently assigned to that process and cancels any reservations for that process.

The reservation of resources and cancellation of reservations are done from command level via the reserve_resource and cancel_resource commands or by using the -resource control argument with the enter_abs_request command. The explicit assignment and unassignment of devices is done from command level via the assign_resource and unassign_resource commands. The listing of reservations, assignments, and attachments is done from command level via the list_resources command. The other commands named here are described in the Commands manual.

Resource Reservation

Users may reserve resources by scheduling with RCP to obtain exclusive rights to a resource for a period of time. RCP enables users to reserve resources or groups of resources through the use of the reserve_resource command (see the Commands manual). A reservation takes effect immediately and it lasts until either the user's process is terminated or the reservation is specifically cancelled via the cancel_resource command (see the Commands manual). After invoking reserve_resource, the user has exclusive rights to the resource(s).

Tape volumes, tape drives, disk volumes, and disk drives can be reserved. Tape and disk volumes are specified at the time of reservation by name; tape and disk drives are specified by either name or attributes. In the case of disk drives, the only acceptable attribute is model. For tape drives, acceptable attributes are model, track, and density. Suitable values for the above-mentioned attributes may be found by using the list_resource_types command (see the Commands manual).

To cancel reservations, users invoke the list_resources command to obtain the reservation identifier, and then invoke the cancel_resource command with the reservation identifier to effect the cancellation. Administrators can perform privileged cancellations; that is, if the administrator has proper access, it is possible to cancel reservations belonging to other users.

Device Assignment

The RCP interface for device assignment allows the caller to request the assignment of a specific device, or any appropriate device of a specified type. To request the assignment of a specific device the caller must ask for the device by name. To request the assignment of an appropriate device of a specified type, the caller must specify the characteristics that the assigned device must have. RCP selects a device for assignment based on the following functional algorithm.

1. If the caller has requested a device by name and if this device is already assigned to the calling process, the assignment is aborted.

2. RCP tests all of the devices of the specified type. RCP counts the number of these devices that are appropriate; appropriate and accessible; and appropriate, accessible and available. These requirements are discussed below:

    a. appropriate: A device is considered to be appropriate if it has the device characteristics specified by the caller. In testing each device, RCP does not try to match any device characteristics that are not specified by the caller. If a device is asked for by name, only the device name is considered.

    b. accessible: A device is considered to be accessible if the calling process has rw RCP effective access to the device.

    c. available: A device is considered to be available for assignment if it is not currently assigned to any process or reserved by another process.

3. Having tested each of these requirements, RCP then makes additional tests to see if a device can be assigned. If the assignment cannot be made, RCP returns an error_table_ code that tells the caller why the assignment aborted. The tests that RCP makes at this time are described below:

   a. If there are no appropriate devices, the caller is told that the requested resource (device) is not known to RCP.

   b. If there are no appropriate and accessible devices, the caller is told that he does not have access to the requested resource (device).

   c. If there are no appropriate, accessible and available devices, the caller is told that the requested resource (device) is not available at this time.

   d. If this assignment causes the device limits (see "Device Limits" below) to be exceeded, the user is told that he has exceeded the limit.

4. If all the tests described above are passed successfully, the device assignment is made. RCP selects the most advantageous device from the list of devices that were found to be appropriate and accessible and available. It makes this selection based on the following rules:

   a. If this is a type of device that has volumes and if the caller specified a volume name to use in the device selection and if any device in the list currently has that volume mounted, RCP selects that device.

   b. If the first case is not true, RCP selects the device that has been idle for the longest amount of time.

Having assigned the device, RCP returns all of the characteristics of this device to the caller.

Device Attachment

The RCP interface for device attachment allows the caller to request a device in the same manner described for device assignment. It can ask for a specific device by name or it can ask for any appropriate device of a specified type. One difference is that if this device is a type that uses volumes, the caller must specify the name of the volume to attach. For assignments, the specification of a volume is optional.

RCP tests all of the devices of the specified type that are already assigned by the requesting process. If the specific device or any appropriate device is already assigned to this process, RCP attaches that device. If no suitable device is already assigned to the requesting process, RCP automatically attempts to assign a suitable device to this process. If no device can be assigned then the attachment is aborted. If the attachment is for a device type that uses volumes, RCP checks to see if the specified volume is already attached to this process or any other process. If the volume is already attached, RCP aborts the attachment.

Once RCP has found a suitable assigned device or has assigned one, it begins the real work of attaching the device. This involves calling IOI to perform the ring 0 device attachment. If the device is a type that uses volumes, RCP tells the operator to mount the specified volume if it is not already mounted on the proper device. Before the attachment is completed, RCP makes sure that the proper volume has been mounted and that any write protection mechanism provided by the device is set correctly. When all of this initialization work has been completed, RCP calls IOI to set the workspace and time-out limits and to promote the validation level of the device. Until this is done, the IOI validation level for the device is the RCP validation level (ring 1). Thus no program in a higher ring can successfully call IOI to use this device until RCP tells IOI to promote it. RCP returns all of the device characteristics of the attached device and all of the information needed to communicate with IOI about this device.

DEVICE LIMITS

In addition to controlling which processes may have access to a device, RCP will enforce a limit to the number of devices of a given type that a single process may have assigned at one time. This limit is enforced according to the following rules:

1. The limit is not enforced for system processes.

2. The limit for each device type is an installation-defined value. It is currently specified in the RTDT.

3. Currently, only tape drive devices actually have such a limit defined.

RESOURCE NAMING CONVENTIONS

While the RCP implementation allows resource names to be any ASCII string of up to 32 characters, there are restrictions placed on some of these names by other sources. Details of these resource naming conventions are described below.

Device Names

Each device has a unique name. Device names are of one of the following forms:

```
ssss_xx
dddd
```

The first form is used for devices that share multiplexed I/O channels such as disk and tape devices. The latter is used for all other devices.

In the case of disk and tape device names the name is composed of the subsystem name, 'ssss' in the text above, and the device number, 'xx' above. The subsystem name is defined by the site in the configuration via a PRPH card (see MOH) and the device number is assigned by the Field Engineering Representative when the hardware is installed.

All other devices are also defined in the configuration deck. In this case the PRPH card defines the device itself. These device types include: consoles, printers, card readers, card punches, and special devices.

The four character restrictions listed above are due to the fact that character fields on configuration cards are limited to four characters (one word).

Volume Names

Volume names are unique within their volume type (i.e., no two tape volumes may have the same name). They may be up to 32 characters in length. The only reserved volume names are "scratch" and "T&D_Volume" which are used to designate scratch volumes for disk and tape. A scratch tape is one of the unmarked tapes in an unreserved pool that is used for "scratch"--that is, no information is saved on it from session to session. After every use, it is demounted and returned to the system pool. "T&D_Volume" is used for special label processing for online Test and Diagnostics, and its use for attachments requires special privilege.

I/O WORKSPACES

Due to the nature of the Multics virtual memory and its supporting I/O hardware, I/O operations such as "read tape" or "write disk" require all pages of memory referenced by the I/O operation to be in main memory during the operation--that is, no paging is done during execution of the I/O operation. To accomplish this, all channel programs and physical record buffer areas are located in a special segment known as an I/O workspace segment. The ring 0 I/O software, IOI, guarantees that all pages of the workspace are present in main memory before starting the I/O operation and remain there for the duration of the operation.

RCP will control the maximum workspace size associated with each device type. System processes, privileged processes, and users on the ACL of the ACS named workspace.acs in the directory >system_control_1>rcp can request up to the privileged maximum workspace size. All others can request up to the normal maximum workspace size. Requests for a workspace larger than is allowed result in errors. The table below lists the workpace maximums that are enforced.

Table 5-7. I/O Workspaces

	Privileged Maximum		Normal Maximum	
device type	words	bytes	words	bytes
tape_drive	45056	180224	6144	24576
disk_drive	45056	180224	2048	8192
printer	45056	180224	1024	4096
punch	45056	180224	1024	4096
reader	45056	180224	1024	4096
special	45056	180224	1024	4096
console	45056	180224	1024	4096

The workspace size is affected by using the -block control argument to those I/O modules that support it. This control argument is used to specify the maximum physical record/block size to be processed. In all cases some overhead for channel programs and I/O module control information must be taken into consideration. When -block is not specified or supported the individual I/O modules choose an appropriate default. In the case of commands that use I/O modules, either the command, some argument or input to the command, or the I/O module may specify/imply in some way the workspace size (for example by supplying -block in an attach description).

RESOURCE MANAGEMENT FACILITY

The Resource Management Facility handles registration and acquisition of resources, which includes release and deregistration. Resource management is a site option, which must be enabled by a system administrator.

RCP software reserves, assigns, and mounts resources; it also demounts, unassigns, and cancels reservations. The RCP is an integral part of Multics.

The hierarchical level of these functions are:

```
1   register
                                  Resource Management
    2   acquire
**************************************************
        3   reserve

            4   assign

                5   attach
                                  Resource Control
                5   detach

            4   unassign

        3   cancel

**************************************************
    2   release
                                  Resource Management
1   deregister
```

Resource management is an optional facility which offers the ability to retain registration information for all resources that it controls. It does this by providing administrative interfaces for the registration of resources (see the Administration, Maintenance, and Operations Commands manual) Registration of a resource provides information such as: what type of resource this is, what its name is, which attributes it possesses, or in what access class range the resource can be used. Once a resource is registered, users may acquire it; system administrators can also acquire it to a user (or to the system pool) at the time it is registered (described in the *Administration, Maintenance, and Operations Commands* manual). The act of acquisition makes a user the owner of the resource--liable for all charges to that resource and in control of discretionary access to the resource.

Summary of Resource Management Facility Actions

Described below are actions that apply when the Resource Management Facility is enabled. When resource management is not enabled, all volumes and devices are effectively acquired to the system.

ACQUIRING RESOURCES

When a system administrator registers a resource, he may simultaneously cause it to be acquired; that is, designate who will become the accounting owner of that resource.

Once a resource is registered by the system administrator, it may be acquired by a user. When a user acquires a resource, he is contracting with the system to become the accounting owner of the resource. In other words, the person who acquires the resource usually agrees to pay a fee for the right to control the access to that resource.

After registering a resource, the system administrator may acquire it in the name of the system or a user, deciding who is allowed to use that resource. Devices (such as tape drives and printers) and "scratch" volumes (e.g., tapes in the system pool) are usually acquired to the system. System-owned resources such as devices and scratch volumes are for use by all users. For other resources such as tape reels and disk packs, the system administrator normally chooses to leave most of these in an unacquired state so that users may acquire these resources on an individual basis.

Once a resource has been acquired it can be used (reserved, assigned, and attached) by any user with appropriate access. See "Access Control" below. Any resource that is not resident in the system or free pools is acquired by a user_id (Person_id.Project_id).

It is important to realize that there is normally no implicit acquisition, and that only acquired resources can be used. The only exception to these rules occurs when a site has "automatic registration" turned on during the initial time period after enabling the full Resource Management Facility. While automatic registration is on, any unregistered tape volume for which the operator honors a mount request is automatically registered and acquired to the requesting user.

In order to control the operation of the Resource Management Facility, an administrative table exists that can be adapted to the specific needs of a particular Multics site. This table is referred to as the resource type description table (RTDT). The table is generated from a source language description, called the resource type master file (RTMF), ordinarily prepared by a system administrator. The contents of the RTDT can be examined via the display_rtdt command. (The RTDT and the display_rtdt command are described in detail in the *Administration, Maintenance, and Operations Commands* manual, Order. No. GB64.

NAMING RULES FOR ATTRIBUTES

Attributes provide a description of a volume or device that assists the Resource Management Facility in the proper matching of volumes with compatible devices. To produce correct combinations, attribute names must comply with the set of rules described below.

Attributes may be grouped or ungrouped. Grouped attributes specify a set of properties applicable to a device or volume such that only one attribute of that set can be currently active at any given time. For example, a reel of tape may have potential attributes that allow it to be recorded at densities of 556, 800, or 1600; however, at any given time, the data on it is in only one of those densities. Grouped attributes have names of the form:

```
<identifier>=<value>
```

For example, the attributes mentioned above are named "den=556", "den=800", and "den=1600". This notation allows RCP to recognize that any request to make one of these attributes the current attribute of a device or volume also implies that all other attributes in that grouping must be made inactive.

When adding or changing a string of attributes, all attributes in the string must be respecified or else exiting attributes are nullified by the change. Also, any attribute string must contain a value for each grouped attribute. For example, if the attribute domain includes "track=... and model=...," the device you are setting the attributes for (or registration) must contain values for each grouped attribute.

Ungrouped attributes have simple names, such as "trainok" (to specify that this device accepts a removable print train) or "building_12" (to specify that this device or volume is located in building 12).

ACCESS CONTROL INTERFACE WITH RCP AND RESOURCE MANAGEMENT

There are three types of access control on Multics: discretionary access control, which is regulated by access control lists (ACL); nondiscretionary access control, which is regulated by the access isolation mechanism (AIM); and intraprocess access control, which is regulated by the ring structure. (For detailed information on types of access, see Section 6.) Access control works differently with and without resource management. These differences are noted in the discussions below.

Access Control Segments

An important feature of RCP is its ability to control access to the various resources that it manages. It does this through the use of access control segments (ACSs). An ACS is a zero length segment whose ACL and ring brackets are used to define the discretionary access to a resource. RCP uses an ACS for each resource that it controls; however, an ACS can be shared by more than one resource. The name of an ACS consists of a name plus the suffix, acs (e.g., tape_01.acs). There are no restrictions on ACS names other than the required suffix. The user creates an ACS and generates/manipulates its ACL with the create, set_acl, and delete_acl commands and ring brackets with the set_ring_brackets command (see the Commands manual).

The pathname of the ACS for a resource is usually specified when it is acquired (see the register_resource command and the acquire_resource command in the Commands manual). The specified ACS can later be changed or unspecified so that the resource (again) has no ACS via the set_resource command (see the Commands manual). If the ACS has not been specified or does not exist, access is by default rew for the owner of the resource and null for all other users.

When resource management is *not* enabled, ACSs exist only for devices, *not* for volumes. These ACSs are automatically created with pathnames of the form:

```
>sc1>rcp>resource_name.acs
```

These pathnames cannot be changed. Access to volumes is determined by site policy.

With resource management enabled, RCP uses the ACS along with other nondiscretionary controls (AIM) to determine the RCP effective access to a resource.

Access Class Ranges

Access class ranges are used by RCP to specify that a process within a range of authorizations can use a particular resource. This discussion pertains to sites where resource management is enabled.

An access class range is simply a pair of AIM access classes separated by a colon. The first value of the pair is the minimum access class and the second is the maximum access class. If only a single access class is specified when an access class range is expected, the minimum and maximum access class values are both the same (i.e., a range of one value). The second access class of the pair (the maximum) must be greater than or equal to the first (the minimum) according to the aim_check_ subroutine (see the Subroutines manual).

There are some interesting results which occur when categories are used in an access class range. For example, a process with authorization of:

```
level2,category1
```

would not be able to use a resource whose access class range was:

```
level1,category1,category2:level3,category1,category2,category3
```

where level3 is greater than level2, which is greater than level1. This is due to the fact that the authorization of the process is isolated from the minimum of the access class range. In order to allow this process access to the resource in question, the range would have to exclude category2 or the user would have to have category2 authorization. In general, to include categories within an access class range, both the minimum and maximum must include the categories desired. If combinations of categories are desired, the minimum should list only required categories and the maximum should include all categories allowed. For example, the access class range:

```
level1,category1:level3,category1,category2,category3
```

allows read and write access to any level1, level2, or level3 process with category1 and any combination of category2 and category3.

RCP Effective Access

Viewed separately, each type of access control answers the same question, "What access does a particular process have for a particular item?" The access mode granted a process to a resource by discretionary access control (the ACL) is known as the raw access mode.

The way RCP determines effective access to a resource for a process differs from the regular Multics method of determining effective access as follows. First, the effective access to the ACS for the resource is determined as for any segment. If the ACS does not exist, the user appears to have read, execute, and write access if he is the owner of the resource, or null access if he is not the owner. Then, two further checks are made. First, the current authorization of the process is compared to the maximum access class of the resource. If write access is not allowed (as defined by the write_allowed_ subroutine) then write and execute access are denied and only read is allowed. Next, the current authorization of the process is compared to the minimum access class of the resource. If read access is not allowed (as defined by the read_allowed_ subroutine) then all access is denied. The resulting access is termed the RCP effective access to the resource. One final restriction enforced by RCP is that, in order to use a device, the RCP effective access must include both read and write to that device (a restriction not imposed on volumes).

For example, the following table illustrates some examples of RCP effective access. In the examples below, L1, L2, L3 and L4 represent sensitivity levels and c1, c2, c3, and c4 represent categories. (This discussion mostly concerns devices--volumes should never be given multiclassed access class range.)

Table 5-8. RCP Effective Access

Effective Access to ACS	Current Process Authorization	Resource Access Class Range	RCP Effective Access
rew	L1	L1:L3	rew
re	L1	L1:L3	re
rew	L1	L2:L3	null
rew	L3	L2:L3	rew
rw	L4	L2:L3	r
re	L4	L2:L3	r
rw	L2,c1	L1:L4	r
rw	L2,c2	L1,c1:L4,c1,c2	null
rw	L2,c1,c3	L1,c1:L4,c1,c2	r
rw	L2,c1	L1,c1:L4,c1,c2	rw

A user must have write RCP effective access to the resource named to perform any modification on the status of the resource. In addition, the user must have execute effective access to the resource named to modify protected attributes. Only the accounting owner may modify the ACS path.

For more information on AIM, access classes, authorizations, and comparisons involving access classes and authorizations, see Section 6. The access class range mentioned above is specified by the -access_class control argument, which can be specified in the register_resource command (see the *Administration, Maintenance, and Operations Commands* manual, Order. No. GB64), and the acquire_resource and set_resource commands (described in the Commands manual).

Manipulating RCP Effective Access

Since the access control mechanisms described above operate together to determine the RCP effective access of a process, there are actions that the user, as well as an administrator, can perform to control this effective access. If resource management is not enabled, however, only the administrator can control access.

First, the user creates an ACS via the create command. Then, the desired ACL for that segment is established using the set_acl command to add desired ACL entries, and the delete_acl command to delete entries. (The above three commands are described in the Commands manual.) To further affect the ACS, the user may modify its ring brackets by using the set_ring_brackets command (described in the Commands manual). The system security administrator sets the AIM access class range of the resource itself at the time it is registered using the register_resource command and can change it by using the set_resource command.

SECTION 6

MULTICS SECURITY

Multics provides a set of complementary data security mechanisms designed to restrict unauthorized access to programs and data. Each set of security mechanisms implements a different level of protection and "defends" the system against different penetration strategies. The use of any one or all of the several mechanisms is optional. When more than one mechanism is used, access is limited to that granted by all controls.

The security mechanisms available on Multics are listed below:

- User Names and Passwords
- Access Control Lists
- Access Isolation Mechanism
- Ring Mechanism

Additionally, Multics provides a "trusted path" connection to the operating system. A trusted path is a guaranteed direct connection between a user at a terminal and the Multics operating system and is designed to protect users against the possibility of their logging in to a simulated system created by a subverter.

USER NAMES AND PASSWORDS

A user name and password must be supplied each time a user logs in to Multics. If the user name and/or password is not supplied correctly, the user is denied access to the system.

At the time a user is registered, an administrator assigns a unique (to Multics) identifying name. Although unique, user names are considered public and thus provide only limited protection against unauthorized use.

At the time a user is registered, the administrator also assigns a password of up to eight characters. The first time the user logs in, the administrator-assigned password should be changed by the user to a new value (known only to the user). Since the password is stored in an irreversibly encrypted file, the user-selected password is completely private. The password mechanism thus provides more complete protection against use of the system by unauthorized individuals.

To take full advantage of the password mechanism, users should take care not to use short passwords and not to use easily decoded values (your first name, your telephone number, etc.). A good idea is to insert a special character (dollar sign, question mark, etc.) in the middle of the password. The use of special characters provides added protection against the possibility of another individual guessing the password.

ACCESS CONTROL LISTS

The access control list mechanism enables users to control access to objects that they own. For each object to be protected, the owner identifies which other users are permitted access and the specific kind of access each is allowed. Individual users can only perform those operations specifically permitted by the owner of the object.

Objects Subject to Access Control

There are four types of objects subject to access control:

1. entries in the storage system (segments, directories, etc.)
2. resources protected by the Resource Control Package (RCP)
3. communications channels
4. daemon source names

The methods used to set access on each of the above objects is described later in this section.

Access Identifier

In order to grant individual users distinct access rights, it is necessary to be able to identify the different users. For this purpose, each user has an associated name called an access identifier. The identifier is a three-component character string that must be less than or equal to 32 characters. The first component is the registered name of the person (i.e., the user's Person_id); the second component is the name of a project group of which the person (named in the first component) is a member (i.e., the user's Project_id); and the third component (called the instance tag) is a single character used to distinguish different classes of processes. Most processes have an instance tag of "a" to indicate a standard interactive process (i.e., a process created for a user who logged in from a terminal). Absentee processes (i.e., noninteractive processes created by the system in response to queued user requests), have an instance tag "m". The instance tag "p" identifies a process entered as a proxy by some user other than the name indicated by the Person_id.Project_id. The instance tag of "z" is used for daemon processes (e.g., one that runs a line printer). The instance tag "o" is applied to access identifiers used to control operator access to daemon source names. The access identifier Jones.Mentor.a would be associated with an interactive process created for Jones on the Mentor project.

It is important to note that if a user is not specifically granted access to an object, then the user cannot access the object in any way.

The access identifier is considered a "user" by the system. However, it is important to distinguish between a user and a person: the same person can log into Multics under two different projects and be considered two different users (e.g., Jones.Mentor.a and Jones.Demo.a), or one person could log in interactively and be running an absentee process at the same time and be considered two different users (e.g., Jones.Mentor.a and Jones.Mentor.m). If a person on a particular project is granted the ability to log in more than once so that he has several processes under his control at the same time, each process has the same access identifier (e.g., Jones.Mentor.a and Jones.Mentor.a). These processes, by having the same access identifier, have the same access rights to segments and directories in the storage system.

Access Modes

User's access rights are described by access modes. Access modes define the kind of operations a user can perform on a specified object. For example, a user who must be able to read data from a segment can be assigned "read access" mode to the segment. A user who must be able to delete entries in a directory can be assigned "modify access" mode to the directory.

There are a variety of access modes corresponding to the different operations that can be performed on the several objects. The various access modes are meaningful only when considered with the associated object. A complete description of the individual access modes that can be applied to each object is provided below.

ACCESS MODES ON ENTRIES IN THE STORAGE SYSTEM

The access modes to be applied to the various entries in the storage system are listed below. Note that, in addition to the access modes specified below, an access mode of n (null) can be set on any storage system entity. Null access mode specifies that the user cannot access the entity in any way.

The access modes for segments are:

r (read)	The user can read data from the segment.
e (execute)	The user can transfer control to this segment and instructions in the segment can be executed on behalf of the user.
w (write)	The user can write data in the segment.

The access modes for directories are:

s (status)	The attributes of entries cataloged in the directory and certain attributes of the directory itself can be obtained by the user (for a definition of attributes, see Section 2).
m (modify)	The attributes of existing entries cataloged in the directory and certain attributes of the directory itself can be modified; and existing entries contained in the directory can be deleted.
a (append)	New entries can be created in the directory.

If n (null) mode is set on a directory, the contents of the directory cannot be read or modified. However, the user can access any entity in the directory to which (s)he has non-null access.

The access modes for multisegment files are:

r (read)	The user can read data from the multisegment file.
w (write)	The user can write data from the multisegment file.

The access modes for data management files are:

r (read)	The user can read data from the data management file.
w (write)	The user can write data from the data management file.

Access modes are not applied to link entries.

The access modes for mailboxes and message segments are listed below. In the discussion below, note that mailboxes contain messages, while message segments contain queued requests (as from the enter_output_request, dprint, and dpunch commands).

a (add)	The user can add a message/request.
d (delete)	The user can delete any message/request.
r (read)	The user can read any message/request.
o (own)	The user can read or delete only his own messages/requests; that is, those sent with the same Person_id.

s (status)	The user can obtain message/request counts.
n (null)	The user cannot access the mailbox/message segment in any way.
w (wakeup)	The user can send an interactive message to the mailbox. This access type is used by the send_message command and related commands (described in the Commands manual). This access type is not available for message segments.

Access on a newly created mailbox is automatically set to adrosw for the user who created it, aow for *.SysDaemon.*, and aow for *.*.*. Access on I/O message segments is controlled by the site, but is usually set to adros for the user who created it, adros for *.SysDaemon.* and aros for *.*.* as a default.

The access modes for forum meetings are:

r (read)	The user can read transactions in the forum meeting.
w (write)	The user can write transactions in the forum meeting.
c (chairman)	The user is the chairman of the meeting and has access to chairman's commands.

The access modes for before journal files are:

r (read)	The user can read data in the before journal.
w (write)	The user can write data in the before journal.

The access modes for the person name table are:

r (read)	The user can read data in the person name table.
w (write)	The user can write data in the person name table.

ACCESS MODES ON RESOURCES PROTECTED BY RCP

An RCP resource is a device or volume that is under management and control of the resource control package (RCP) facility.

The access modes for RCP volumes are:

r (read)	The user can read data on the volume.
e (executive)	The user can act as an executive for a storage system logical volume.
w (write)	The user can write data on the device or volume.

n (null) The user cannot access the volume in any way.

The access modes for RCP devices are:

rw (read/write) The user can use this device (reserve, assign, or attach)

ACCESS MODES ON COMMUNICATIONS CHANNELS

The access modes for communications channels are:

rw (read/write) The user can attach this channel (subject to restrictions in the CMF)

ACCESS MODES ON DAEMON SOURCE NAMES

The access modes for daemon source names are:

r (reply) The user is permitted to execute the initializer reply command.

q (quit) The user is permitted to execute the initializer quit command.

c (control) The user is permitted to login/logout the specified daemon.

d (daemon) The user is permitted to login using the specified source name.

Creating, Modifying, Listing, and Deleting Items in an Access Control List

A user can create an entry in an access control list by means of the set_acl command.[1] The set_acl command requires the user to identify:

1. the object (segment, directory, etc.) to be protected,
2. the individual to be granted access rights, and
3. the kind of access each individual is allowed.

---

[1] The access control list for RCP resources, communications channels, and daemon source names is set on a particular entity called an access control segment. The access control segment must first be created by the user before the set_acl command can be used to create entries in an access control list. See "Access Control Segments" later in this section.

For example, if user Smith on the Sales project were to be granted read access to the Accounts segment (located at >udd>Records>Accounts), the set_acl command would be specified as follows:

```
sa >udd>Records>Accounts r Smith.Sales
```

The list_acl command displays the access control list for a specified object. To obtain the access control list associated with the segment named Employees (located at >udd>Personnel>Employees), the list_acl command is specified as follows:

```
la >udd>Personnel>Employees
```

The delete_acl command removes items from the access control list of a specified object. To remove user Smith on the Sales project from the access control list associated with the segment Accounts (located at >udd>Records>Accounts), the delete_acl command is specified as follows:

```
delete_acl >udd>Records>Accounts Smith.Sales
```

Granting Access to Groups of Individuals

When granting access to groups of individuals, the following conventions are used:

1. as asterisk (*) is used to replace one (or more) of the components of an access identifier
2. one of the access identifier components is not specified.

USING THE ASTERISK CHARACTER

An asterisk character (*) can replace one, two, or all of the three components of an access identifier. The asterisk character is a "wild card" entry and specifies that *any* value in that position is a valid value.

The asterisk character is useful in granting access to groups of users. For example, to grant all individuals in the Sales project read access to the Accounting segment (located at >udd>Records>Accounting), the set_acl command is specified as follows:

```
sa >udd>Records>Accounts r *.Sales
```

This method eliminates necessity for specifying an acl entry for each individual on the Sales project, as in:

```
sa >udd>Records>Accounts r Able.Sales r Baker.Sales r Charles.Sales...
```

Similarly, if a user were registered on several projects, the asterisk value could be used in place of the project component to grant access to the user regardless of project value. For example, use of the identifier Jones.* would give user Jones the specified access regardless of whether Jones were logged in on the project Maintenance (Jones.Maintenance) or the project Development (Jones.Development).

The ultimate use of the asterisk convention is to use it in place of all three component values of the access identifier (*.*.*), thus granting access to everyone on the system.

NOTE: Because all other access identifiers take precedence over *.*.*, it is more accurate to say that the *.*.* convention grants access to everyone who is not also the subject of another, more specific acl entry. See "Calculating Access Rights" below for additional information.

MISSING COMPONENTS

A missing component is treated as if an asterisk were used. For example Smith.Sales is the same as Smith.Sales.* and Smith is the same as Smith.*.*. Missing components on the right need not be delimited by periods. Missing components on the left must be delimited by periods. If an access identifier is specified as .Sales, then Sales is interpreted by the system as the project_id component (the second component). If an access identifier is specified as ..a, then a is interpreted by the system as the instance tag (the third component).

Calculating Access Rights

The system places each access identifier in a particular position in the access control list according to the following rules:

```
Position  Component
```

1. access identifiers with no asterisks
2. access identifiers with an asterisk in the third component only
3. access identifiers with an asterisk in the second component only
4. access identifiers with asterisks in the second and third components only
5. access identifiers with an asterisk in the first component only
6. access identifiers with asterisks in the first and third components only

7. access identifiers with asterisks in the first and second components only

8. access identifiers with all asterisks (*. *. *)

Thus, the following is an example of an ordered access control list:

```
Smith.Multics.a       r
Jones.Multics.a       null
Smith.*.*             rw
*.Multics.*           re
*.*.z                 rw
*.*.*                 r
```

When the system searches the list to calculate access, the first matching identifier encountered determines the access rights.

Thus, in the acl list specified above, user Smith logged in to project Multics as an interactive user (a) is given only read access to the segment. However, if user Smith is logged in on any other project but Multics, then user Smith is able to obtain read/write access. Everyone on the Multics project is granted read/execute access but Smith and Jones. All daemons (z) get read/write access.

Note that the last item gives everyone read access to the segment. However, because of the way the list is ordered and searched, exceptions must be made. For example, user Jones gets no access at all. Conversely, other users obtain more general access. User Smith (on all projects but Multics) gets read/write access; individuals on the Multics project get read/execute access.

It is important to remember, then, that an identifier granting access to groups of individuals (say, *.*.*) does not necessarily grant access to *all* members of the group. Individuals in the group may be identified in other entries in the same access control list. The system grants access as specified by the *first* identifier that matches the name of the searching individual.

Initial ACL's

Each time a storage system entry is created, the system automatically enters certain "initial acl's" into the acl list. The system default is to enter an entry for the SysDaemon project and an entry for the user creating the segment or directory. Additionally, users can specify their own "initial acl " list by means of the set_iacl_dir and set_iacl_seg commands.

SYSDAEMON ENTRIES

Multics provides service routines (daemons) that perform functions such as making backup copies of segments in the storage system and printing and punching segments at users' requests. In order to perform such functions, the service routines must have access to the segments to be serviced. The service routines (and only the service routines) are members of a single project called SysDaemon.

In order to ensure that daemons have access to the segments, the system automatically places the ACL entry:

rw *.SysDaemon.*

on the ACL of every segment, and the ACL entry:

sma *.SysDaemon.*

on the ACL of every directory when the segment or directory is created or its ACL is entirely replaced. In this way, members of the SysDaemon project are automatically granted the necessary access so that they can perform their functions; individual users need not remember to put the proper entries on all of their segment and directory ACLs to make use of the daemon processes.

Under special circumstances, some user might not wish to use the facilities of a daemon on some segments. In this case, the user simply denies that daemons access to the segments by modifying the ACL entry (i.e., giving that daemon null access). It is crucial that a user who elects not to use a daemon be fully aware of the nature of the service and the consequences of the choice. For example, if the hierarchy backup daemons are not permitted access to a segment, backup copies of the segment cannot be made and the segment will not survive certain types of system failure.

ACL ENTRY FOR THE CREATING USER

In addition to automatically adding a daemon entry to the ACLs of all newly created storage system entries, many system commands and subroutines (e.g., create, create_dir, and hcs_$append_branch), add an entry for the creating user to the ACL of a newly created segment or directory. For a data segment, that ACL entry is:

```
rw   Person_id.Project_id.*
```

For an object segment, the ACL entry is:

```
re   Person_id.Project_id.*
```

Note that for both the daemon entry and the creating user entry, the instance tag is designated by an asterisk, meaning that otherwise matching process identifiers have access to these segments regardless of which of the four instance tags they have.

USER-DEFINED INITIAL ACL'S

For convenience, the system allows a user to specify a list of entries to be added to all newly created storage system entries--in addition to entries for the daemons and for the creating user. This ability eliminates the need to explicitly modify an ACL each time a new entry is created.

The set_iacl_dir command permits the user to specify the ACL entries to be automatically placed on all newly-created directories within the specified directory. For example, to specify that all individuals in the Multics project are to be given status access to the directories within the Records directory, the set_iacl_dir command is specified as follows:

```
sid >udd>Records s *.Multics.*
```

The set_iacl_seg command permits the user to specify the ACL entries to be automatically placed on all newly-created segments within a specified directory. For example, to specify that everyone is to be given read/write/execute access to segments within the working directory the set_iacl_seg command is specified as follows:

```
sis -wd rew *.*.*
```

Access Control Segments

An access control segment is a zero length segment that is associated with the object to be protected. A user's access to the access control segment determines the user's ability to perform operations on an RCP resource, communications channel, or daemon.

ACCESS CONTROL SEGMENTS FOR RCP RESOURCES

The access control segments for RCP resources are handled differently depending on whether or not the resource management option (RCRPM) is enabled.

When the resource management facility of RCP *is not* enabled, an access control segment can exist for devices only (not for volumes). The access control segment must have the form <resource_name>.acs (e.g., tape_01.acs). These access control segments are automatically created and stored in the directory >sc1>rcp.

When the resource management facility of RCP *is* enabled, an access control segment can exist for devices and volumes. The access control segment must be created using the create command. The segment name must end with the suffix .acs; the segment can exist any place in the hierarchy.

ACCESS CONTROL SEGMENTS FOR COMMUNICATIONS CHANNELS

The access control segment for communications channels can be created by an administrator using the create command. The access control segment must be in the directory >sc1>rcp. The segment name must be in the form <channel_name>.acs.

ACCESS CONTROL SEGMENTS FOR DAEMON SOURCE NAMES

An access control segment can be associated with daemon source names (the source_id used to login the daemon). By setting appropriate access rights on the ACS, an administrator can control operator access to the daemon. The ACS must be located in the directory >sc1>mc_acs and must be of the form >sc1>mc_acs>SOURCE_NAME.mcas, where SOURCE_NAME is the source identifier of the daemon. Complete information on the setting of access controls for daemon source names is contained in the *Multics System Administration Procedures Manual*, Order No.: AK50.

ACCESS ISOLATION MECHANISM

The Access Isolation Mechanism (AIM) is a security mechanism that allows the separation of data into different levels of privilege and controls the flow of information across the different levels. AIM is an administrative tool and is implemented on a site-wide basis (in contrast to access control lists which are implemented by individuals to protect individual data files).

Use of AIM involves (1) marking each object with an "access class" and (2) marking each user with an "authorization." AIM determines information access on the basis of the access class of the object and the authorization of the user.

AIM Classification System

Multics AIM uses a classification system known in the literature as a "lattice model." In this model, users and objects are marked with two items:

1. category (kind, or type of data) and/or
2. sensitivity level.

For example, a company may wish to divide its data into the following categories: Personnel, Marketing, and Engineering. Second, the company could recognize that data is subject to different levels of sensitivity; for example, the information could be classed as public, proprietary, or confidential.

Authorization information composed of the same category/level attributes is maintained for each user in the system. When a user references a piece of data, the system determines what access will be granted based on the category/sensitivity level of the data and the category/sensitivity level of the user.

Policy Rules and Objectives

There are two basic rules that make up the Multics security policy.

1. No information can flow from a higher (more sensitive) level to a lower (less sensitive) level.

2. No information can flow between category boundaries.

There are two reasons to isolate users and their data from other users and their data.

1. To prevent owners of information from granting access to it inappropriately or unintentionally.

2. To deal effectively with the "trojan horse" problem

Trojan Horses are programs that exploit the access privileges of the program to make data available to other users. Say, for an example, that a programmer is given the task of writing a program that will be used to analyze some proprietary information. The programmer can exploit this fact by adding code to the analysis program to copy the protected information into one of his segments. Or, the programmer can act to change the ACL on the data, though that leaves more obvious evidence.

In another example, a subverter could obtain a game program and insert new code to copy all of the game user's segments. The game user would only "see" the game interface while the trojan horse code secretly copied the user's data segments.

Relationships Between AIM Attributes

The AIM access rules (described in "AIM Access Rules" below) are based on the following relationships between authorizations and access classes:

```
equal to
greater than
less than
isolated from
```

An authorization or access class A is equal to an authorization or access class B if:

1. The sensitivity levels of A and B are equal; and

2. The category sets of A and B are identical (neither contains a category not found in the other).

An authorization or access class A is greater than an authorization or access class B if:

1. The sensitivity level of A is greater than or equal to the sensitivity level of B; and

2. The category set of B is a subset of the category set of A or is identical to the category set of A; and

3. A is not equal to B (according to the above definition of equal to).

An authorization or access class A is less than an authorization or access class B if B is greater than A.

An authorization or access class A is said to be isolated from authorization or access class B if A is not equal to, greater than, or less than B. Two authorizations with the same level can be isolated from one another only if neither's category set is equal to or a subset of the other. (An empty category set is considered a subset of all nonempty category sets.)

Setting AIM Attributes

The information below describes the system tools for setting AIM attributes. Administrators should have available a precisely-defined set of rules for information transfer before enabling the AIM security mechanisms.

ENABLING AIM

The system administrator enables the AIM mechanism by specifying values for one or more of the following keywords in the ed_installation_parms statement:

```
access_ceiling
category_names
level_names
```

The category_names key word defines the number and identity of the different access categories. The level_names keyword defines the number and identity of the different sensitivity levels (level_names). The access_ceiling keyword defines the maximum sensitivity level that can be used and the total number of categories that can be used.

The sensitivity levels and access categories used in a particular Multics installation are assigned character-string names for convenience.. There may be as many as eight different sensitivity levels and 18 access categories in use at one Multics installation. If an installation has chosen not to use the AIM access controls, that system is using only the lowest sensitivity level and no categories. The access classes and authorization names at such an installation are null strings by default, making access classes and authorizations "invisible."

MARKING OF DATA

Data can be "marked" with only one sensitivity level. The sensitivity level is a site-defined value (e.g., L1 or L2 or L3) conveying a relative sensitivity judgement. If the contents of segment A are judged to be more sensitive than the contents of segment B, segment A should have a higher sensitivity level than segment B.

Data can be "marked" as belonging to one or more categories. The category is a site-defined value (e.g. C1, C2, C3) representing a grouping of information. The list of categories assigned forms a "category set." The administrator should note that, if data is marked as belonging to several categories, only users authorized for *all* categories will be allowed to access the data.

The assignment of level/category information to elements in the file system is described below. The term "access class" is used to refer to the combined level/category information.

Segment

A segment receives its access class, equal to the access class of the containing directory, at the time it is created. No special commands or control arguments are needed to assign access classes to segments.

Directory

Like a segment, a directory receives its access class at the time of creation. If no access class is explicitly requested, the directory is assigned an access class equal to the access class of its containing directory. If an access class is explicitly requested, it must be greater than or equal to the access class of the containing directory and less than or equal to the process maximum authorization. A directory with an access class higher than that of its containing directory is an upgraded directory. No directory may have an access class less than that of its containing directory, so the access class of directories always remains the same or increases as one descends the hierarchy. This is known as the "compatibility" rule.

An upgraded directory must be explicitly assigned storage system quota of one or more storage records. Quota may be moved to an upgraded directory from its containing directory by a process whose authorization is equal to the access class of the containing directory. Quota may not be moved from an upgraded directory back to its containing directory except by deleting the upgraded directory. An upgraded directory may be deleted only if it is empty (contains neither segments nor links).

To explicitly assign access attributes to a directory, use the -access_class control argument to the create_dir command, as in the following example.

```
cd DirA -access_class L1,C1,C2 -quota 5
```

The example above creates the directory DirA with an access class of L1,C1,C2 (where L1 is a sensitivity level value as specified for the site and C1 and C2 are categories, as specified for the site). A -quota control arguments is required; the example sets the quota value at 5 pages.

Message Segment

A single message segment can contain messages of different access classes. The access class of each message in the segment is equal to the authorization of the user that added the message to the message segment.

The access class of the message segment itself controls the maximum access class of the messages in it. Every message must have an access class less than or equal to the access class of the message segment.

The access class of a message segment is equal to the maximum authorization of the user that created it.

Mailboxes

The read_mail and send_mail commands (described in the Commands manual) use message segments to hold mail, each piece of mail being a single message with an access class equal to the authorization of the sending user. AIM access controls impose several restrictions on the use of mail. Since a user can read only messages with access classes less than or equal to his authorization, a user cannot read mail sent by users of higher authorizations. Since a user can only delete messages with access classes equal to his authorization, he cannot delete mail with an access class not equal to his authorization.

The access class of a mailbox is equal to the maximum authorization of the user that created it. Since all messages in a mailbox must have an access class less than or equal to the access class of the mailbox, a user can read all his mail when his authorization is equal to his maximum authorization. However, he may not be able to delete all his mail at this authorization. In general, mail is easiest to manage if it is only sent and read by users at one authorization. In this case, a user can read and delete all of his mail. Users wishing to send and read mail of multiple authorizations may experience the inconveniences of having messages in their mailbox that they cannot read or delete at certain authorizations.

MARKING OF USERS

Individual users are "marked" by sensitivity level and category.

The sensitivity level is a site-defined value (e.g., L1 or L2 or L3) conveying a relative sensitivity judgement. If user A is more trusted than user B, then user A should have a higher sensitivity level than user B.

The category is a site-defined value (e.g., C1, C2, C3) representing a grouping of information. The list of categories assigned to a user forms a category set. A user with several categories is authorized to access several information groups. (The administrator should note that, if data is marked as belonging to several categories, only users that are authorized for all the categories will be allowed access to the data.)

The marking carried by any individual user depends on the authorization values (sensitivity level and category) placed in the following three system tables: the System Administrator Table (SAT), the Person Name Table (PNT), and the Project Definition Table (PDT).

A new user is registered on the system by means of the new_user command. The new_user command permits the administrator to specify minimum and maximum authorization values. The minimum and maximum values represent the range of authorizations permitted for the specified individual. The authorization range is entered in the PNT.

A new project is registered on the system by means of the new_proj command. The new_proj command permits the administrator to specify minimum and maximum authorization values. The minimum and maximum values represent the range of authorizations permitted for individuals on the specified project. The authorization range is entered in the SAT and the PDT.

A project administrator can subsequently change the authorization values (for an individual or for the project) in the PDT. The values in the PDT, however, cannot be outside of the maximum and minimum values specified in the SAT.

At the time that a user logs on, the user's authorization is set to that maintained in the PNT. (If, however, the authorization range maintained in the SAT or PDT is more restricted, then the user's authorization is set to the more restricted range.) At login time, the user can elect to use the -auth argument to override the values maintained in the PNT. (However, the value specified for the -auth argument cannot be outside the most restricted of the values maintained in the PNT, the SAT, or the PDT.) It should also be noted that an authorization range can be specified for the communications channel used to login. If the user's authorization range is not within the range specified for the communications channel, the user will not be permitted to login over the channel.

MARKING OF RCP RESOURCES

The system administrator can enable the resource control package (RCP) resource management facility to manage the use of peripheral I/O devices (such as tape drives, and disk drives) and physical volumes that can be mounted on these devices (such as tape reels and disk packs). The resource control package permits the administrator to assign access class ranges to the device/volumes.

Access class ranges are used by RCP to specify that a user within a range of authorizations can use a particular resource.

An access class range is simply a pair of AIM access classes separated by a colon. The first value of the pair is the minimum access class and the second is the maximum access class. If only a single access class is specified when an access class range is expected, the minimum and maximum access class values are both the same (i.e., a range of one value). The second access class of the pair (the maximum) must be greater than or equal to the first (the minimum).

The user should be aware of results which occur when categories are used in an access class range. For example, a process with authorization of:

```
level2,category1
```

would not be able to use a resource whose access class range was:

```
level1,category1,category2:level3,category1,category2,category3
```

where level3 is greater than level2, which is greater than level1. This is due to the fact that the authorization of the process is isolated from the minimum of the access class range. In order to allow this process access to the resource in question, the range would have to exclude category2 or the user would have to have category2 authorization. In general, to include categories within an access class range, both the minimum and maximum must include the categories desired. If combinations of categories are desired, the minimum should list only required categories and the maximum should include all categories allowed. For example, the access class range:

```
level1,category1:level3,category1,category2,category3
```

allows read and write access to any level1, level2, or level3 process with category1 and any combination of category2 and category3.

The administrator uses the access_range parameter in the Resource Type Master File (RTMF) to specify an access class range for a given resource. (See the *System Administration Procedures* manual for additional information on RCP.)

MARKING OF COMMUNICATION CHANNELS

The system administrator can assign an authorization range to a specified communications channel. The administrator uses the access_class statement in the channel master file to specify the authorization range.

The authorization can be specified as a single value, in which case the channel is usable only by users with the specified authorization. The authorization can be specified by a minimum and maximum value, in which case the channel is usable only by users with an authorization equal to or greater than the minimum value and equal to or less than the maximum value.

If the access class statement is not specified, the value is assumed to be that specified (or defaulted to) by the Access_class global statement.

The administrator must be aware that the system cannot establish the authorization of a user for any channel except channels identified as multiplexer_type sty. For this reason, it is recommended that all channels except those identified as multiplexer_type sty should be specified with a single access_class value. The only exception to this recommendation is for login service type channels. These channels should be assigned an authorization range sufficient to cover the users who are to be permitted to log in over the channel. (See the *System Administration Procedures* manual for additional information on the marking of communication channels.)

AIM Access Rules

The access rules used by AIM on segments, directories, interprocess communication, and message segments are described below.

SEGMENTS

The rules for accessing segments are:

1. A user may have read (r) and execute (e) modes to a segment only if the user's authorization is greater than or equal to the segment access class.

2. A user may have write (w) mode to a segment only if the user's authorization is equal to or less than the segment access class.

   NOTE: Write mode permission allows the user only to append information (it does not allow the user to read, modify, or delete existing data).

3. A user has null access to a segment if its authorization is neither greater than nor equal to the segment access class.

4. A user may have read/write access to a segment only if its authorization is equal to the segment access class.

DIRECTORIES

The rules for accessing directories are:

1. A user may have status (s) mode to a directory only if the user's authorization is greater than or equal to the directory access class.

2. A user may have modify (m) and append (a) modes to a directory only if the user's authorization is equal to the directory access class.

3. A user has null access to a directory if its authorization is neither greater than nor equal to the directory access class.

A newly created segment has the same access class as its containing directory. A newly created directory may have an access class that is greater than or equal to the access class of its containing directory. A directory with an access class greater than its containing directory is known as an upgraded directory.

MESSAGE SEGMENTS

A message segment is a special type of segment that is managed by Multics supervisor programs and is not directly accessible to the user. A message segment is simply a convenient repository for interprocess messages. Each message in a message segment is a separate protection unit itself, and has associated with it an access class identical in form to segment and directory access classes. The existence of the individual messages remains invisible to a process unless the process authorization is greater than or equal to the message access class. A process may read a message only if the process authorization is greater than or equal to the access class of the message. A process may delete messages only if the process authorization is equal to the message access class. A process may get the count of messages in a message segment, but this count only reflects those messages to which read access is permitted by AIM.

INTERPROCESS COMMUNICATION

The interprocess communication (IPC) facility allows one process to pass information to another process by sending it a wakeup and an associated event message. Administrative access controls limit the use of this information path. Process A may send a wakeup (and event message) to process B only if process B's authorization is greater than or equal to process A's authorization.

Inter-System AIM

Facilities like the Inter-Multics File Transfer (IMFT) Facility translate AIM attributes between two systems. For these facilities, the concept of a common access class ceiling is used to control the data which may be transferred between the systems.

The common class ceiling between two systems is defined as:

- all sensitivity levels from level 0 (usually unnamed) up to but not including the first level which does not have the same long and short name on both systems, and
- all access categories that have the same long and short names on both systems

If the long and short names of sensitivity level 0 are not the same on both systems, then the two systems have no common access ceiling and are isolated from each other.

For example, if system A defines the following AIM attributes.

```
level 0       (unnamed)
level 1       unclassified          u
level 2       secret                s
level 3       top secret            ts

category 1    Personnel             pers
category 2    Planning              plng
category 3    Finance               (none)
category 4    Marketing             (none)
```

and system B defines the following attributes:

```
level 0       (unnamed)
level 1       unclassified          u
level 2       restricted            (none)

category 1    Engineering           (none)
category 2    Planning              plng
category 3    Finance               fin
category 4    Personnel             pers
```

then the common access ceiling is:

```
unclassified, Planning, Personnel
```

THE RING MECHANISM

All data and executable code on Multics resides within a logical entity called a "ring." A ring is a conceptual structure that confers a particular level of privilege on the information that is within the ring.

There are eight rings (0 through 7) on Multics. Ring 0 is the ring of most privilege. Ring 7 is the ring of least privilege.

The prime rule of access between rings is that code executing in lower-numbered rings has unlimited access to data in higher-numbered rings (subject, of course, to ACL and AIM restrictions). Code executing is higher-numbered rings has no direct access to data in lower-numbered rings. (Access refers to the ability to execute code as well as the ability to read and write data.)

Advantages of the Ring Mechanism

To ensure proper operation, the operating system software must be protected from accidental or intentional user modification. However, although the operating system software must be protected, it cannot simply be made inaccessible. Users must frequently call on the code in the operating system to perform some function on their behalf. The ACL and AIM mechanisms are not adequate security mechanisms in this circumstance. With ACL's and AIM, it is not possible to restrict what the user does with the data beyond the basic restrictions of reading, writing, and executing. The ring mechanism, however, makes it possible to grant access to a user, but only to perform some specified, approved procedure.

Ring Attributes and Access Control

All segments (and directories) in the storage system possess ring attributes. The ring attributes are a series of three numbers (in the range of 0 through 7). For example, the ring attributes assigned to a particular segment might be expressed as [6,6,6], [2,5,6], or [0,0,4].

Each user is assigned a particular ring to which he is initially assigned at login and in which he can begin to execute code. If the user's initial ring value is 6, then the user is logged in within ring 6. If a user's initial ring value is 4, then the user is logged in within ring 4.

RING BRACKETS

The three ring values assigned to each segment determine the segment's "ring brackets." Ring brackets are access brackets that specify the read, write, execute, and gate access.

Write Bracket

The rings less than or equal to the first of the ring bracket numbers are termed the write bracket. A user must be executing in a ring within the write bracket of a segment and have write access mode on that segment in order to modify data on that segment. If a user is running in a ring higher than the write bracket, the user cannot modify (write into) the segment even though the user has write access.

Read Bracket

The rings less than or equal to the second ring bracket number are called the read bracket. Users must be running in the read bracket of a segment and have read access in order to read it.

Execute Bracket

The first and second ring numbers are used to determine the execute bracket. Users must be running in the execute bracket of a segment in order to execute code in the segment.

There are two subsets of the execute bracket.

1. If a user is executing in a ring whose number is less than the first ring number, then the segment can be executed (if the user has execute access). However, the user's ring of execution will be changed to the lowest ring in the segment's execute bracket.

   NOTE: An attempt to execute code residing in a higher ring number is termed an "outward call". A outward call will succeed only if (1) no arguments are passed, and (2) the code to be executed resides within the highest ring number in which the calling process is permitted to run. If the call violates either of these two conditions, an "outward call condition" is generated.

2. If a user is executing in a ring whose number is the same as or is between the first and second ring bracket numbers, then the segment can be executed (if the user has execute access) without having to change the ring of execution of the user.

For example, the ring bracket number assigned to a segment are [3,5,6], execute access is determined as follows:

1. If the user is executing in ring 0, 1, or 2, then the user has execute access. However, upon transfer to the segment, the user's ring of execution will be (temporarily) changed to 3.

2. If the user is executing in ring 3, 4, or 5, then the user has execute access. The user's ring of execution remains at 3, 4, or 5.

Gate Bracket

The third number in the segment attribute series determines a gate bracket. Rings greater than the second ring bracket number and less than or equal to the third ring bracket number are within the gate bracket.

The gate bracket (or gate) is a means of making segments in the inner rings accessible to the segments in the outer rings, but in a controlled manner. The inner ring procedure segment that is specified as a "gate" usually contains executable code that performs a special function and then returns control to the user at a point outside the gate ring. Upon transfer to the gate, the user's ring of execution changes to that of the gate segment. The third ring number must be specified at least one greater than the second ring number in order for the segment to qualify as a gate. Upon transfer to the gate, the users ring of execution changes it that of the gate segment.

NULL ACCESS

If the user's ring of execution is greater than the third ring bracket number (whether specified or defaulted to), the user has no access to the segment.

Using the Ring Mechanism

The power of the ring structure lies in the use of the execute and gate brackets. These rings allow users to define arbitrary procedures and then encapsulate these procedures in a closed, controlled environment that can be entered only at specified gate entry points. Some operating system segments, for example, would have small read, write, and execute brackets, but large gate brackets. This would make the procedure accessible to a wide variety of users, but accessible only under carefully controlled circumstances.

The following example illustrates use of the gate mechanism.

Suppose a user executing a program in ring 6 references in turn segments A, B, C and D, which have, respectively, ring numbers [6,6,6], [4,4,6], [2,5,6] and [0,0,4] and that the AIM and ACL mechanisms allow the user execute access to all these segments.

In the course of executing segment A, the process calls segment B. Since segment A is in a ring outside the execute bracket for B, but within its gate bracket, it is granted access to B and its current ring number becomes 4. In the course of executing segment B, it calls segment C. Since it is within the execute bracket for segment C, it is granted access and its ring number remains the same. In the course of executing segment C, it calls segment D. Since it is within the gate bracket of segment D, it is granted access, and its current ring number becomes 0. When it finishes executing D, it is automatically returned first to segment C in ring 4, then to segment B in ring 4, and then to segment A in the ring in which it began, ring 6. This process is illustrated in Figure 6-1. Note that the process cannot call segment D from segment A and that it cannot skip the intermediate gate, B, and still reach the ring 0 segment D by calling C from A and D from C. Note also that the process is admitted to ring 0 only through the gate segment D. The only functions that can be performed in ring 0 are those included in segment D. Segment D should be coded to perform an arbitrary (limited) procedure and then return the process to a point outside ring 0. This example may suggest how the ring mechanism gives administrators the ability to determine the circumstances under which a sequence of segments can be called.

In addition to protecting the operating system, the ring mechanism is used to protect user subsystems. For example, a teacher could restrict students to ring 5 by asking a system administrator to allow users on the teacher's project to log in only in ring 5. The teacher might then write a gate segment with ring numbers [4,4,5] and an ACL granting execute access to all users on the project, and a grade-book segment with ring numbers [4,4,4] and an ACL granting write access to all users on the project. When the students finished homework problems in a segment in ring 5, they could call the teacher's gate into ring 4. The gate segment would examine the student's work, store a grade in behalf of the student in the grade-book segment, and return to the student in ring 5. Because the students would have access to the grade-book segment only through the gate, they would not be able to examine or modify the grades. The teacher, who could log in on ring 4, however, would.

This process is illustrated in Figure 6-2.

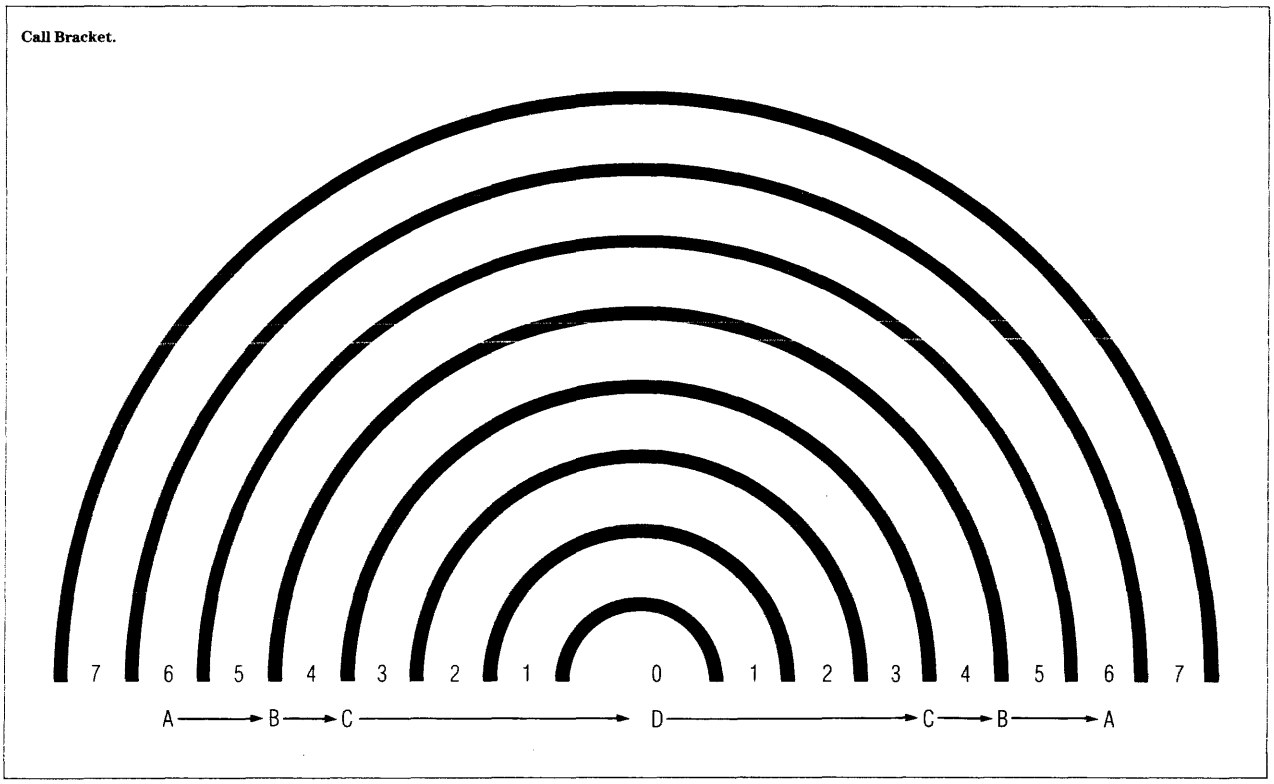


Figure 6-1. Gate Mechanism

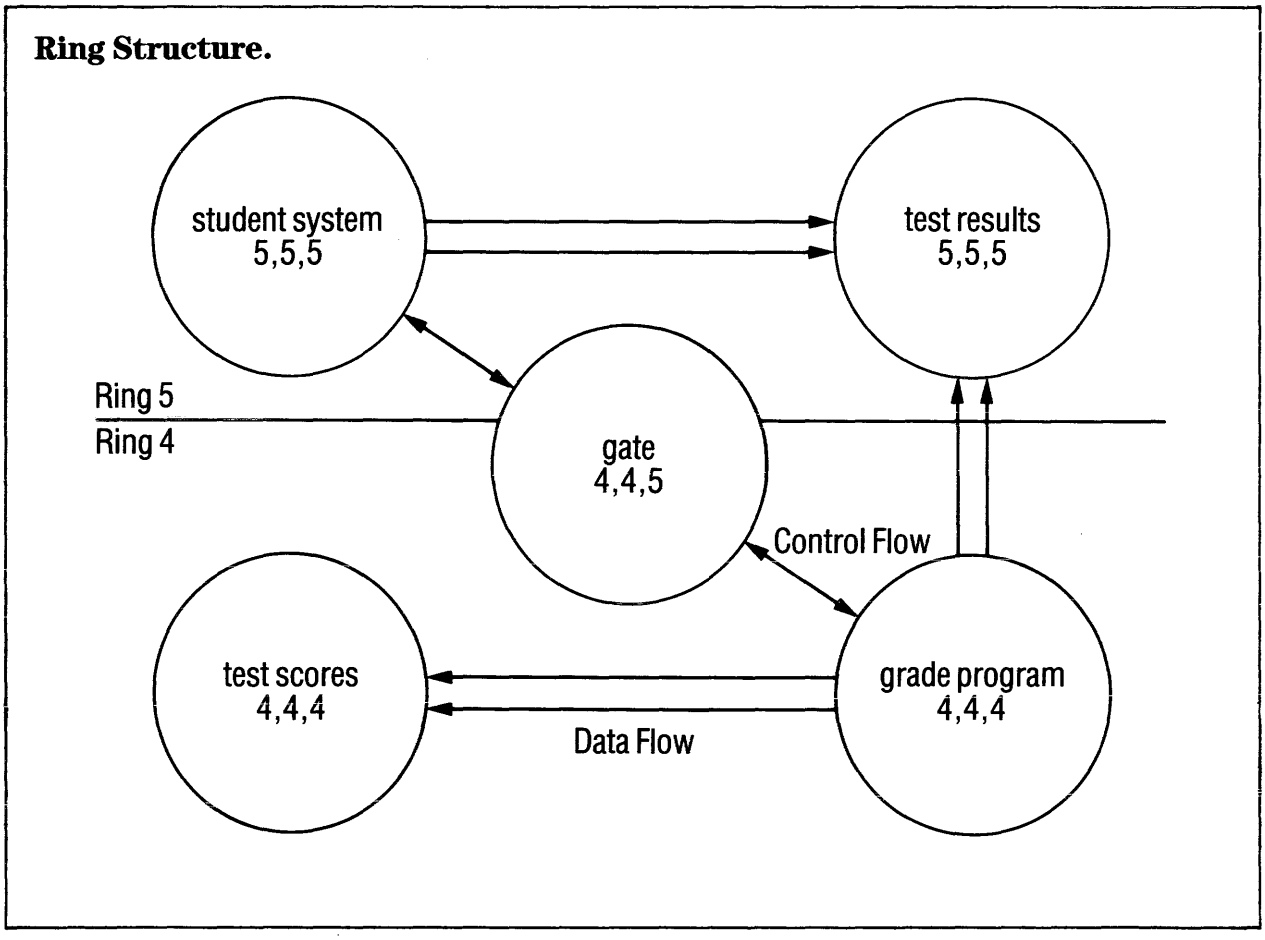


Figure 6-2. Logical Flow in Homework Program

Implementing Ring Protection

The ring mechanism is an integral part of the Multics system. All operating system code runs in protected rings, as determined by system programmers. All registered users are placed within a particular ring, as determined by the site administrator. Administrators and users can make use of the ring mechanism as desired.

In a typical system, for example, rings 0 through 3 would be reserved for operating system use. Most user processes would start running in ring 4. Rings 5 through 7 would be used by programmers to write their own protected subsystems (as previously described in the "homework" problem).

SETTING SEGMENT RING BRACKETS

All segments within the storage system possess ring attributes (a three-number ring bracket value).

The -ring_brackets control argument to the create (segment) command allows the user to set the ring brackets for the segment. The user need not explicitly specify any ring brackets. In that event, the system assumes a default value for the brackets (as explained below).

The user can specify one, two, or three ring bracket numbers. (If the user specifies only one number, the system assumes it is the first ring bracket number; if the user specifies two ring bracket numbers, the system assumes that they are the first and second ring bracket numbers.) In the event the user specifies only one ring bracket number, the system assumes assumes a default value for the other two. The default value for the other two is the value of the first ring number. In the event the user specifies two ring bracket numbers, the system assumes a default value for the third. The default value for the third is the same as the second. In the event the user does not use the -ring_bracket control argument (specifies no ring bracket numbers), the value of the the ring brackets becomes [x,x,x], where x is the user's ring number as established at login.

The ring bracket numbers must also be specified according to the following rules

1. The second ring bracket number must be equal to or greater than the first

2. The third ring bracket number must be equal to or greater than the second

3. To be considered a gate, the third ring bracket number must be at least one greater than the second.

For example, to create a segment with ring brackets 2, 5, and 6, the user would type:

```
create test.prog1 -ring_brackets 2 5 6
```

To create a segment with ring brackets of 6, 6, and 6, the user need type:

```
create test.prog2 -ring_brackets 6
```

If the user logged in to ring 5 and wanted to create a segment with the ring brackets 5, 5, 5, the user need only type:

```
create testprog3
```

Note that in the first example above, the segment test.prog1 is a gate since the third number is greater than the second.

The system prevents gate access from being established across system projects. Unless the operation is performed from the ring-1 administrative ring or by a system daemon, no user can create a gate that is accessible by anybody who does not belong to that user's own project. This restriction ensures that a user on one project cannot construct a gate that allows, for example, ring-5 users on another project to access restricted ring-4 data in a way not allowed by the gate provided them by their project administrator.

MODIFYING SEGMENT RING BRACKETS

The user can modify the ring brackets of a previously created segment by means of the set_ring_brackets command.

For example, to modify the ring brackets of an existing segment (testprog2) to [2,4,6], type:

```
srb test.prog2 2 4 5
```

The rules for specifying the ring bracket numbers in the set_ring_brackets command are the same as those specified for the create command. See "Setting Segment Ring Brackets" for detailed information.

DIRECTORY RING BRACKET VALIDATION LEVEL AND ACCESS RIGHTS

Directory ring brackets control modifications to attributes and directory contents. Directory ring brackets do *not* use the ring of execution; they use the "validation level."

Validation Level

Inner ring procedures are very often called by outer ring procedures in order to perform some service on behalf of the outer ring. It is, therefore, necessary that the inner ring procedure know the number of the outer ring on whose behalf it is performing the service in order to validate the right of the outer ring to request the service. This requesting ring information is kept by each process and is known as the *validation level*. If an outer ring procedure wishes to request a service from an inner ring procedure, it sets the validation level to its current ring of execution (the validation level cannot be set lower than the ring of execution) and calls the inner ring procedure. If a procedure is calling an inner ring procedure to do work on behalf of an outer ring procedure, it should not change the validation level, but instead leave it at the level of the outer ring procedure. Users who write programs that are executed only in a single ring, usually the outermost ring in which the process runs, need not be concerned about the validation level since it will be set to that ring by default.

Directory Ring Bracket Access Rights

Directory ring bracket access rights differ from those of segments due to the following factors:

1. There are only two directory ring brackets, not three.

2. Since directories are accessed by calling supervisor primitives rather than by direct reference, the directory ring brackets are evaluated with respect to the validation level instead of the ring of execution.

The first ring bracket number defines the *modify/append bracket*. All rings less than or equal to the first directory ring bracket number are within the modify/append bracket. In order for a user to modify or add entries to a directory, the validation level must be within the modify/append bracket and the user must have modify or append access modes (respectively) on the directory. The rings less than or equal to the second directory ring bracket number form the *status bracket*. In order to get the attributes of segments in a directory or of inferior directories, the validation level must be within the status bracket. The first ring bracket number must be less than or equal to the second ring bracket number. For example, if the ring brackets of a directory are 4,6 and the validation level is 3, the user can get status of, modify, or append to the directory (assuming, of course, that he has the status, modify, and append access modes). If the validation level is 6, the user can only get status of the directory. If the validation level is 7, the user cannot access the attributes of the entries in the directory at all.

SETTING DIRECTORY RING BRACKETS

When a directory is created, ring bracket numbers need not be explicitly specified, in which case the ring brackets are set to the current validation level.

When a directory is created, ring bracket numbers can be explicitly specified using the -ring_brackets control argument to the create directory command.

For example, to set ring brackets of [4,6] on the directory >udd>Engin>ProjA, the user need type:

```
cd >udd>Engin>ProjA -ring_brackets 4 6
```

MODIFYING DIRECTORY RING BRACKETS

The user can modify the ring brackets of a previously-created directory by means of the set_dir_ring_brackets command.

For example, to modify the ring brackets of the previously-created directory >udd>Engin>ProjA to [5,7], type:

```
sdrb >udd>Engin>ProjA 5 7
```

USER RING BRACKETS

At the time a project is registered, the administrator sets (either explicitly or by default) the per-project ring attributes for all users registered on the project. Individual users can be assigned per-user values; however the per-user values cannot provide more access (to either higher or lower numbered segments) than that specified by the per-project values.

The per-project and per-user ring attributes include (1) the lowest ring in which a user can create a process, (2) the highest ring in which a user can create a process, and (3) the ring in which the user is placed at login (this value must lie in the range defined by the lowest/highest ring attribute values).

The per-user values need not be specified. If not specified, the system uses the per-project values. The per-project values need not be specified. If not specified, the system uses a default value of 4 for the lowest ring value and a default value of 5 for the maximum ring value.

The table below describes a user's access rights to a segment from the perspective of that users current ring of execution (ring in which the user is executing a program).

Ring of Execution	*Potential Access Rights*
Ring of execution less than first ring bracket number	read, write, execute (with ring change)
Ring of execution equal to first ring bracket number	read, write, execute
Ring of execution greater than first ring bracket number and less than or equal to second ring bracket number	read, execute
Ring of execution greater than second ring bracket number and less than or equal to third ring bracket number	execute (if a gate only, with ring change)
Ring of execution greater than third ring bracket number	no access

TRUSTED PATH

A "trusted path" is a guaranteed direct connection between a user at a terminal and the Multics operating system. The trusted path mechanism is designed to protect users against the possibility of their logging in to a simulated system created by a subverter. A trusted path should be available for all requests signalling a change in the process environment (new password, new authorization, new process, etc.).

To obtain a trusted path connection between your terminal and the Multics operating system, you must cause a terminal "hangup" condition. This will, in turn, cause the answering service (a Multics system process) to monitor the line. When you reconnect (login), you can be assured you are communicating with Multics operating system software.

In order to cause a "terminal hangup condition" you must cause your communications line to drop the Data Terminal Ready (DTR) signal. This can be accomplished in one of several ways:

1. Physically turning off your terminal.

2. Breaking the connection between your terminal or modem and the communications line.

3. Causing your terminal to drop the DTR signal automatically. Some terminals have this capability. (On Honeywell VIP 7800 series terminals, you place the terminal in "local" mode, and, while holding down the CTL key, typing P, D, RESET).

Once the DTR signal is dropped, you should ensure that the Multics Front-End Processor (FNP) has detected the loss of the DTR signal. It indicates this by dropping the Data Set Ready (DSR) signal. Some terminals, modems, or multiplexers allow you to monitor this signal. You should verify that the DSR signal has been dropped. (On Honeywell VIP 7800 series terminals, the Data Set Ready light will be go out).

Once you have verified that DSR has been inhibited, perform whatever action is necessary to reinstate the DTR signal. Several possibilities include:

1. Turning the terminal back on, if it was turned off.

2. Reconnecting the terminal or modem to the communications line, possibly re-dialing the phone number for Multics.

3. Causing your terminal to signal DTR again. (On Honeywell terminals, this is accomplished simply by taking the terminal out of "local" mode.

The Multics FNP will signal its receipt of DTR by reapplying DSR. This may cause the DSR indicator on your terminal or modem to light up again. The Multics answering service (a Multics system process) will then monitor the line, and shortly thereafter, the Multics banner will appear. When you login again, you can be assured you are communicating with the Multics operating system.

Note that this procedure assumes your communications hardware allows you to drop the DTR signal and that this loss of DTR is reflected all the way to the Multics FNP. (Some networks will prevent Multics from seeing the loss of DTR). In addition, some way of sensing the loss of DSR is required as well.

SECTION 7

HANDLING UNUSUAL OCCURRENCES

A program may encounter a set of circumstances that prevent it from continuing normally. Examples of circumstances that prevent a program from continuing execution are an attempt to divide by zero or the inability to find a necessary segment in the storage system. Clearly, whether or not a particular set of circumstances, such as those given above, prohibit a program from continuing in a normal manner is dependent upon the program in question. Circumstances that are abnormal for one program can be quite normal when encountered in a different program. If a program is unable to continue, it notifies its caller or other of its antecedents. The handling of such occurrences and the notification mechanisms are described in this section.

The discussion is limited to methods of handling unusual occurrences reported by system programs. However, it should help users select appropriate means for handling and reporting unusual occurrences that arise during the execution of their own programs. Printed messages, status codes, conditions, and faults are discussed.

PRINTED MESSAGES

The type of unusual occurrence reporting that most Multics users first encounter is a message printed on the user's terminal. Since, in some sense, the caller of a command is the user himself, printing a message on the user's terminal is the means by which a command can report an unusual occurrence to its caller. There are two general types of printed messages used to report unusual occurrences: statements and questions. A statement describes the occurrence to the user. The user may then rectify the circumstances by issuing commands. A question describes the occurrence and requests an immediate response from the user in the form of a character string entered at the terminal. In this way, the user must immediately specify one of several courses of action that the command takes with respect to the occurrence.

Most Multics system commands generate printed messages in a standard format. This format consists of the name of the command printing the statement or asking the question and a description of the unusual occurrence and the question. Two procedures, the com_err_ and command_query_ subroutines, are provided to help report unusual occurrences through printed statements and questions. They provide many facilities besides simple formatting. (See the Subroutines manual for descriptions of these subroutines.)

STATUS CODES

Because the character string is too cumbersome for passing descriptions of unusual occurrences between procedures, a coded description of the unusual occurrence, called the status code, is used. The status code is an arithmetic number that takes on a different value for each possible unusual occurrence. The status code argument is passed from a calling procedure to the called procedure. The called procedure assigns the appropriate value to the argument at some point during its execution. When the called procedure returns to the calling procedure, the calling procedure examines the status code to determine what unusual occurrence has been encountered, if any, and then takes special action, if desired. The status code is a means by which a called procedure can report an unusual occurrence only to its immediate caller. However, the first caller may, in turn, pass the status code to its immediate caller, and so on.

The standard status codes used in the Multics system are coded integers. They are referenced by symbolic name (resolved by the linker) and thereby provide a simple means of allowing programs to retain their initial meaning even though the actual representation of a status code may change.

The internal structure of a status code has information allowing subroutines such as com_err_ and convert_status_code_ to acquire a character string description of the status to be printed or returned. Due to the details of the internal representation of the status codes, they are valid only for the process using them and should not be passed from one process to another. However, the system standard table of status codes, error_table_, is valid in all processes.

In order to have a status code generated, a Multics standard status code segment must exist. This segment contains an externally defined symbol corresponding to each status code to be generated in the segment, as well as space for the code itself and the character string interpretation of the code. When the status code segment is first referenced in a process, the system generates a value for each status code defined in the segment and stores it in the segment. From then on, all references to that external symbol refer to the generated status code. The com_err_ subroutine, when given such a status code, is able to locate and return the associated character string interpretation.

A program must refer to a status code symbolically. If, for example, a program wished to return a status code that appears in the status code segment named mistake and has the external symbol bad_argument, then the following PL/I statements would be needed:

```
declare mistake$bad_argument fixed bin(35) external;

return (mistake$bad_argument);
```

If a program wanted to examine a status code for a particular value to determine if it should take some distinct action, it would contain statements such as:

```
declare mistake$bad_argument fixed bin(35) external;

if status_code = mistake$bad_argument then do;
```

All references to the status code are symbolic. The mechanism for generating the status code is automatic and not visible to the program or programmer.

Most Multics system procedures use standard status codes. A list containing the symbolic names, character string interpretations, and meanings of the status codes returned by system procedures is given in "List of System Status Codes and Meanings" below.

Creation of Status Code Tables

Status code tables are constructed using ALM macros which are defined in the include file et_macros.incl.alm. See the description of the alm command in the Commands manual.

Each status code is defined by the ec macro that has as arguments the name, short message, and long message associated with the code. Any number of names may be given to a status code; each name must be 31 characters or less. Multiple names must be separated by commas and the list enclosed in parentheses.

The short message is 8 characters or less in length. If it is omitted, it is set to the code name. The terminating comma must not be omitted.

The long message is 100 characters or less in length and is enclosed in parentheses.

The macro's argument list may contain no blanks except within the long message.

The syntax for the ec macro is:

```
ec code_name,short_message,(long message)
```

The et macro initializes the code table and must appear first and only once in the source. The syntax for the et macro is:

```
et name_of_code_table
```

The following is a sample status code table:

```
include  et_macros
et       user_errors
ec       too_few_arguments,toofew,(There were too few arguments.)
ec       could_not_access_data,noprivig,
         (User is not sufficiently privileged to access data.)
ec       (fatal,disaster),disaster,
         (There was a disastrous error in the data base.)
end
```

Each status code in the table produced by these macros should be referenced as a fixed binary(35) quantity, known externally:

```
declare user_errors$disaster fixed bin(35) external,
      code fixed bin(35);

call data_base_manager (info, code);
if code = user_errors$disaster /* this is bad */
then call kill_subsystem;
```

LIST OF SYSTEM STATUS CODES AND MEANINGS

Status codes report unusual occurrences encountered by procedures during execution. The codes are returned by Multics system commands and subroutines. Printed messages that correspond to these status codes appear on printed output with the name of the command printing the statement, a description of the unusual occurrence causing the message to be printed, and more detailed information when appropriate. The following status codes are all defined in the error_table_ segment and should be referenced as a fixed binary(35) quantity, known externally:

```
declare error_table_ $xxx fixed bin(35) external;
```

abs_reenter:
absentee: Attempt to reenter user environment via a call to cu_$cl. Job terminated.

abs_timer_runout:
absentee: CPU time limit exceeded. Job terminated.

action_not_performed:
The requested action was not performed.

active_function:
This command cannot be invoked as an active function.

ai_above_allowed_max:
Specified access class/authorization is greater than allowed maximum.

ai_invalid_binary:
Unable to convert binary access class/authorization to string.

ai_invalid_range:
The specified access classes/authorizations are not a valid range.

ai_invalid_string:
Unable to convert access class/authorization to binary.
ai_no_common_max:
There are no access classes/authorizations in common between the two systems.
ai_out_range:
The specified access class/authorization is not within the permitted range.
ai_outside_common_range:
The access class/authorization is not within the range in common between the two systems.
ai_restricted:
Improper access class/authorization to perform operation.
already_assigned:
Indicated device assigned to another process.
already_initialized:
Initialization has already been completed and will not be re-done.
apt_full:
Active process table is full. Could not create process.
archive_component_modification:
This procedure may not modify archive components.
archive_fmt_err:
Format error encountered in archive segment.
archive_pathname:
Archive component pathname not permitted.
area_too_small:
Supplied area too small for this request.
arg_ignored:
Argument ignored.
argerr:
There is an inconsistency in arguments to the storage system.
asynch_change:
A previously referenced item has been changed by another opening.
asynch_deletion:
Record located by seek_key has been deleted by another opening.
asynch_insertion:
Record with key for insertion has been added by another opening.
att_loop:
Attachment loop.
bad_acl_mode:
Bad mode specification for ACL.
bad_arg:
Invalid argument.
bad_arg_acc:
Improper access to given argument.
bad_bar_sp:
The signaller could not use the saved sp in the stack base for bar mode.
bad_channel:
Incorrect IO channel specification.
bad_class_def:
Bad class code in definition.
bad_command_name:
Improper syntax in command name.

bad_conversion:
Error in conversion.
bad_date:
The date is incorrect.
bad_day_of_week:
The day-of-the-week is incorrect.
bad_density:
Incorrect recording media density.
bad_dir:
There is an inconsistency in this directory.
bad_entry_point_name:
Illegal entry point name in make_ptr call.
bad_equal_name:
Illegal syntax in equal name.
bad_file:
File is not a structured file or is inconsistent.
bad_first_ref_trap:
Illegal structure provided for trap at first reference.
bad_handler_access:
Improper access on handler for this signal.
bad_index:
Internal index out of bounds.
bad_label:
Incorrect detachable medium label.
bad_link_target_init_info:
Illegal initialization info passed with create-if-not-found link.
bad_link_type:
Illegal type code in type pair block.
bad_linkage_access:
Improper access on user's linkage segment.
bad_mode:
Improper mode specification for this device.
bad_mode_syntax:
Invalid syntax in mode string.
bad_mode_value:
Invalid value for specified mode.
bad_mount_request:
Mount request could not be honored.
bad_mpx_load_data:
Inconsistent multiplexer bootload data supplied.
bad_ms_file:
Directory or link found in multisegment file.
bad_name:
The access name specified has an illegal syntax.
bad_new_key:
Bad argument to specify the new key of a record.
bad_process_type:
Invalid process type.
bad_processid:
Current processid does not match stored value.
bad_ptr:
Argument is not an ITS pointer.

bad_resource_spec:
Resource specification is invalid.
bad_ring_brackets
Validation level not in ring bracket.
bad_segment:
There is an internal inconsistency in the segment.
bad_self_ref:
Illegal self reference type.
bad_stack_access:
Improper access on user's stack.
bad_string:
Unable to process a search rule string.
bad_subr_arg:
Invalid argument to subroutine.
bad_tapeid:
Invalid volume name.
bad_time:
The time is incorrect.
bad_trap_before_link:
Trap-before-link procedure was unable to snap link.
bad_uidpath:
UID path cannot be converted to a pathname.
bad_volid:
Invalid volume identifier.
bad_work_class:
Specified work class is not currently defined.
bad_year:
The year is not part of the 20th Century (1901 through 1999).
badcall:
Procedure called improperly.
badequal:
Illegal use of equals convention.
badopt:
Specified control argument is not accepted.
badpath:
Bad syntax in pathname.
badringno:
Input ring number invalid.
badstar:
Illegal entry name.
badsyntax:
Syntax error in ascii segment.
bdprtdmp:
Bad part dump card in config deck.
begin_block:
Entry is for a begin block.
big_ws_req:
Insufficient access to use specified block size.
bigarg:
Argument too long.

bigger_ext_variable:
External variable or common block is not the same size as other uses of the same name.
bisync_bid_fail:
Bisync line did not respond to line bid sequence.
bisync_block_bad:
Attempt to write improperly formated bisync block.
bisync_reverse_interrupt:
Reverse interrupt detected on bisync line.
blank_tape:
The rest of the tape is blank.
boundviol:
Attempt to access beyond end of segment.
buffer_big:
Specified buffer size too large.
cannot_trace:
This entry cannot be traced.
change_first:
Attempt to change first pointer.
chars_after_delim:
Segment contains characters after final delimiter.
clnzero:
There was an attempt to move segment to non-zero length entry.
command_line_overflow:
Expanded command line is too large.
copy_sw_on:
There was an attempt to delete a segment whose copy switch was set.
cyclic_syn:
Cyclic synonyms.
data_gain:
Data has been gained.
data_improperly_terminated:
Relevant data terminated improperly.
data_loss:
Data has been lost.
data_seq_error:
Data sequence error.
date_conversion_error:
Unable to convert character date/time to binary.
defs_loop:
Looping searching definitions.
dev_nt_assnd:
IO device not currently assigned.
dev_offset_out_of_bounds:
Specified offset out of bounds for this device.
device_active:
I/O in progress on device.
device_attention:
Condition requiring manual intervention with handler.
device_attention_during_tm:
Device attention condition during eof record write.

device_busy:
The requested device is not available.
device_end:
Physical end of device encountered.
device_limit_exceeded:
The process's limit for this device type is exceeded.
device_not_usable:
Device is not currently usable.
device_parity:
Unrecoverable data-transmission error on physical device.
device_type_unknown:
Device type unknown to the system.
dial_active:
The process is already serving a dial qualifier.
dial_id_busy:
The dial identifier is already in use.
dir_damage:
Directory irreparably damaged.
dirlong:
Directory pathname too long.
dirseg:
This operation is not allowed for a directory.
discrepant_block_count:
Number of blocks read does not agree with recorded block count.
dmpinvld:
Attempt to re-copy an invalid dump.
dmpvalid:
Attempt to modify a valid dump.
dmpr_in_use:
The resource is presently in use by a system dumper.
dt_ambiguous_time:
There is no language common to all words in the time string.
dt_bad_day_of_week:
The date given is not on the indicated day of the week.
dt_bad_dm:
The day of the month is invalid.
dt_bad_dy:
The day of the year is invalid.
dt_bad_format_selector:
The format string contains a selctor which is not defined.
dt_bad_fw:
The fiscal week number is invalid.
dt_bad_my:
The month number is invalid.
dt_conflict:
There is a conflicting combination of day-in-calendar, day-in-year, month-in-year, day-in-month or fiscal-week.
dt_date_not_exist:
The time period 1582-10-05 through 1582-10-14 does not exist.
dt_date_too_big:
The date is after 9999-12-31 GMT.

dt_date_too_small:
The date is before 0001-01-01 GMT.
dt_hour_gt_twelve:
The hour value exceeds 12.
dt_multiple_date_spec:
A date has been given more than once.
dt_multiple_diw_spec:
A day of the week values has been given more than once..
dt_multiple_meaning:
The time string does not have the same meaning in all potential languages, these being the intersection of all the languages possible for all words present.
dt_multiple_time_spec:
A time value has been given more than once.
dt_multiple_zone_spec:
A time zone has been given more than once..
dt_no_format_selector:
The format string contains no selectors and is not a known keyword..
dt_no_interval_units:
No units given in which to express the interval.
dt_offset_too_big_negative:
Applying an offset gives a date before 0001-01-01 GMT.
dt_offset_too_big_positive:
Applying an offset gives a date after 9999-12-31 GMT.
dt_time_conversion_error:
An error has been found while converting a time string. For any of the following reasons:
a. General syntax error
b. Month without a day number.
c. Midnight or noon preceded by an hour other than 12.
d. Improper use of comma or period.
e. Improper use of offset.
dt_unknown_time_language:
The language given is not known to the system.
dt_unknown_word:
An unknown word was found in the time string.
dt_year_too_big:
In the specified time zone the clock value is after the year 9999.
dt_year_too_small:
In the specified time zone the clock value is before the year 0001.
dup_ent_name:
Duplicate entry name in bound segment.
duplicate_file_id:
File identifier already appears in file set.
duplicate_request:
A duplicate request was encountered.
echnego_awaiting_stop_sync:
Echo negotiation race occurred. Report this as a bug.

ect_already_initialized:
The event channel table has already been initialized.
ect_full:
The event channel table was full.
eight_unaligned:
A pointer that must be eight word aligned was not so aligned.
empty_acl:
ACL is empty.
empty_archive:
Archive is empty.
empty_file:
File is empty.
empty_search_list:
Search list is empty.
end_of_info:
End of information reached.
entlong:
Entry name too long.
eof_record:
End-of-file record encountered.
eov_on_write:
Encountered end-of-volume on write.
event_calls_not_masked:
Event calls are not in masked state.
event_channel_cutoff:
Event channels in cutoff state.
event_channel_not_cutoff:
Event channels not in cutoff state.
fatal_error:
A fatal error has occurred.
file_aborted:
Defective file section deleted from file set.
file_already_opened:
File is already opened.
file_busy:
File already busy for other I/O activity.
file_is_full:
There is no more room in the file.
file_not_opened:
No file is open under this reference name.
fim_fault:
Illegal procedure fault in FIM by user's process.
first_reference_trap:
A first reference trap was found on the link target segment.
fnp_down:
The FNP is not running.
force_unassign:
The Operator refused to honor the mount request.
frame_scope_err:
Attempt to reference temporary storage outside the scope of this frame.
full_hashtbl:
The directory hash table is full.
fulldir:
There was an attempt to delete a non-empty directory.

hardcore_sdw:
Attempt to perform an illegal action on a hardcore segment.
higher_inconsistency:
The lock was set on behalf of an operation which must be adjusted.
id_already_exists:
Supplied identifier already exists in data base.
id_not_found:
Supplied identifier not found in data base.
illegal_activation:
There was an illegal attempt to activate a segment.
illegal_deactivation:
There was an illegal attempt to delete an AST entry.
illegal_ft2:
Attempt to indirect through word pair containing a fault tag 2 in the odd word.
illegal_record_size:
Record size must be positive and smaller than a segment
imp_bad_format:
Format of IMP message was incorrect.
imp_bad_status:
Bad status received from IMP.
imp_down:
Multics IMP is down.
imp_rfnm_pending:
A RFNM is pending on this IMP link.
improper_data_format:
Data not in expected format.
improper_termination:
An improper attempt was made to terminate the process.
incompatible_attach:
Attach and open are incompatible.
incompatible_encoding_mode:
Incompatible character encoding mode.
incompatible_file_attribute:
Specified attribute incompatible with file structure.
incompatible_term_type:
The specified terminal type is incompatible with the line type.
inconsistent:
Inconsistent combination of control arguments.
inconsistent_ect:
The event channel table was in an inconsistent state.
inconsistent_msf:
Multisegment file is inconsistent.
inconsistent_rnt:
The reference name table is in an inconsistent state.
inconsistent_sst:
Active Segment Table threads in the SST are inconsistent.
incorrect_access:
Incorrect access to directory containing entry.
incorrect_device_type:
Device type is inappropriate for this request.
incorrect_volume_type:
Volume type is inappropriate for this request.
infcnt_non_zero:
There was an attempt to make a directory unknown that has inferior segments.

insufficient_open:
Insufficient information to open file.
invalid_array_size:
The size of an array passed as an argument is invalid.
invalid_ascii:
The name specified contains non-ascii characters.
invalid_backspace_read:
Invalid backspace_read order call.
invalid_block_length:
Invalid physical block length.
invalid_channel:
The event channel specified is not a valid channel.
invalid_copy:
There was an attempt to create a copy without correct access.
invalid_cseg:
Internal inconsistency in control segment.
invalid_delay_value:
Invalid delay value specified.
invalid_device:
Attempt to attach to an invalid device.
invalid_elsize:
Invalid element size.
invalid_expiration:
File expiration date exceeds that of previous file.
invalid_file_set_format:
File set structure is invalid.
invalid_label_format:
File set contains invalid labels.
invalid_line_type:
Line type number exceeds maximum permitted value.
invalid_lock_reset:
The lock was locked by a process that no longer exists. Therefore the lock was reset.
invalid_max_length:
Attempt to set max length of a segment less than its current length.
invalid_mode:
Invalid mode specified for ACL.
invalid_move_qmax:
Attempt to move more than maximum amount of quota.
invalid_move_quota:
Invalid move of quota would change terminal quota to non terminal.
invalid_mpx_type:
Invalid multiplexer type specified.
invalid_preaccess_command:
Undefined preaccess command.
invalid_project_for_gate:
Invalid project for gate access control list.
invalid_pvtx:
Invalid Physical Volume Table Entry index specified.
invalid_read:
Attempt to read or move read pointer on device which was not attached as readable.
invalid_record_desc:
Invalid variable-length record descriptor.

invalid_record_length:
Invalid logical record length.
invalid_resource_state:
The request is inconsistent with the current state of the resource(s).
invalid_ring_brackets:
The ring brackets specified are invalid.
invalid_seek_last_bound:
Attempt to manipulate last or bound pointers for device that was not attached as writeable.
invalid_setdelim:
Attempt to set delimiters for device while element size is too large to support search.
invalid_stack_creation:
Attempt to create a stack which exists or which is known to process.
invalid_state:
Request is inconsistent with current state of device.
invalid_subsystem:
The specified subsystem either does not exist or is inconsistent.
invalid_system_type:
The specified system type does not exist.
invalid_tp_value:
The supplied value is not acceptable for this tuning parameter.
invalid_volume_sequence:
Specified volumes do not comprise a valid volume set.
invalid_vtoce:
There was an attempt to use a VTOCE with invalid fields.
invalid_vtocx:
The VTOCE index specified is not within the range of valid indices for the device.
invalid_write:
Attempt to write or move write pointer on device which was not attached as writeable.
invalidsegno:
There was an attempt to use an invalid segment number.
io_no_permission:
Process lacks permission to alter device status.
io_still_assnd:
IO device failed to become unassigned.
ioat_err:
Error in internal ioat information.
ioname_not_active:
Ioname not active.
ioname_not_found:
Ioname not found.
ionmat:
Ioname already attached and active.
ips_has_occurred:
An interprocess signal has occurred.
item_too_big:
The item specified is over the legal size.
itt_overflow:
Not enough room in ITT for wakeup.
key_duplication:
There is already a record with the same key.

key_order:
Key out of order.
known_in_other_rings:
There was an attempt to terminate a segment which was known in other rings.
last_reference:
This operation would cause a reference count to vanish.
lesserr:
Too many "<" 's in pathname.
line_status_pending:
Operation not performed because of outstanding line_status information.
link:
This operation is not allowed for a link entry.
linkmoderr:
The execute access is needed to directory containing the link.
lock_is_invalid:
The lock does not belong to an existing process.
lock_not_locked:
Attempt to unlock a lock that was not locked.
lock_wait_time_exceeded:
The lock could not be set in the given time.
locked_by_other_process:
Attempt to unlock a lock which was locked by another process.
locked_by_this_process:
The lock was already locked by this process.
log_vol_full:
The logical volume is full.
logical_volume_is_connected:
The logical volume is already attached.
logical_volume_is_defined:
The logical volume is already mounted.
logical_volume_not_connected:
The logical volume is not attached.
logical_volume_not_defined:
The logical volume is not mounted.
logical_volume_table_full:
The logical volume table is full.
long_record:
Record is too long.
longeql:
Equals convention makes entry name too long.
master_dir:
This operation is not allowed for a master directory.
max_depth_exceeded:
The maximum depth in the storage system hierarchy has been exceeded.
mdc_bad_quota:
Master directory quota must be greater than 0.
mdc_exec_access:
Executive access to logical volume required to perform operation.
mdc_illegal_account:
Illegal format of quota account name.
mdc_mdir_registered:
Quota account has master directories charged against it.
mdc_mdirs_registered:
Volume cannot be deleted because it contains master directories.

mdc_no_access:
Process lacks sufficient access to perform this operation.
mdc_no_account:
Specified quota account not found.
mdc_no_quota:
Insufficient quota on logical volume.
mdc_no_quota_account:
No quota account for the logical volume.
mdc_not_mdir:
This operation allowed only on master directories.
mdc_path_dup:
Pathname already listed.
mdc_path_dup_args:
Pathname appears more than once in the list.
mdc_path_not_found:
Pathname not found.
mdc_path_restrict:
Path violates volume or account pathname restriction.
mdc_some_error:
One or more of the paths given are in error.
mdc_unregistered_mdir:
Master directory missing from MDCS.
media_not_removable:
The specified volume cannot be unloaded from its device.
messages_deferred:
User has deferred messages.
messages_off:
User not accepting messages or not logged in.
mismatched_iter:
Mismatched iteration sets.
missent:
Missing entry in outer module.
mode_string_truncated:
Mode string has been truncated.
moderr:
Incorrect access on entry.
mount_not_ready:
Requested volume not yet mounted.
mount_pending:
Mount request pending.
msf:
This operation is not allowed for a multisegment file.
multiple_io_attachment:
The stream is attached to more than one device.
mylock:
There was an attempt to lock a directory already locked to this process.
name_not_found:
The name was not found.
namedup:
Name duplication.
ncp_error:
Network Control Program encountered a software error.
negative_nelem:
Negative number of elements supplied to data transmission entry.

negative_offset:
Negative offset supplied to data transmission entry.
net_already_icp:
An initial connection is already in progress from this socket.
net_bad_gender:
Bad socket gender involved in this request.
net_fhost_down:
Foreign host is down.
net_fhost_inactive:
Communications with this foreign host not enabled.
net_fimp_down:
Foreign IMP is down.
net_icp_bad_state:
Initial connection socket is in an improper state.
net_icp_error:
A logical error has occurred in initial connection.
net_icp_not_concluded:
The initial connection has not yet been completed.
net_invalid_state:
Request is inconsistent with state of socket.
net_no_connect_permission:
Process lacks permission to initiate Network connections.
net_no_icp:
There is no initial connection in progress from this socket.
net_not_up:
Network Control Program not in operation.
net_rfc_refused:
Request for connection refused by foreign host.
net_socket_closed:
Network connection closed by foreign host.
net_socket_not_found:
Specified socket not found in network data base.
net_table_space:
The NCP could not find a free table entry for this request.
net_timeout:
Connection not completed within specified time interval.
new_offset_negative:
New offset for pointer computed by seek entry is negative.
new_search_list:
A new search list was created.
newnamerr:
User name to be added to acl not acceptable to storage system.
nine_mode_parity:
Attempt to write invalid data in 9 mode.
no_a_permission:
Append permission missing on directory.
no_append:
Append permission missing.
no_archive_for_equal:
No archive name in original pathname corresponding to equal name.
no_backspace:
Requested tape backspace unsuccessful.
no_channel_meters:
No meters available for the specified channel.

no_component:
Component not found in archive.
no_connection:
Unable to complete connection to external device.
no_cpus_online:
The requested group of CPUs contains none which are online.
no_create_copy:
Unable to create a copy.
no_current_record:
There is no current record.
no_defs:
Bad definitions pointer in linkage.
no_delimiter:
No delimiters found in segment to be sorted.
no_device:
No device currently available for attachment.
no_dialok:
The process does not have permission to make dial requests.
noaccess:
Some directory in path specified does not exist.
no_e_permission:
No execute permission on entry.
no_ext_sym:
External symbol not found.
no_file:
File does not exist.
no_fim_flag:
The FIM flag was not set in the preceding stack frame.
no_fnps_configured:
There are no FNPs configured.
no_handler:
No unclaimed signal handler specified for this process.
no_info:
Insufficient access to return any information.
no_initial_string:
No initial string defined for terminal type.
no_io_interrupt:
No interrupt was received on the designated IO channel.
no_iocb:
No I/O switch.
no_key:
No key defined for this operation.
no_label:
Specified detachable volume has no label.
no_line_status:
No line_status information available.
no_linkage:
Linkage section not found.
no_m_permission:
Modify permission missing on entry.
no_makeknown:
Unable to make original segment known.
no_memory_for_scavenge:
Insufficient memory for volume scavenge.

no_message:
Message not found.
no_move:
Unable to move segment because of type, access or quota.
no_next_volume:
Unable to continue processing on next volume.
no_null_refnames:
The segment was not initiated with any null reference names.
no_operation:
Invalid I/O operation.
no_r_permission:
No read permission on entry.
no_record:
Record not located.
no_restart:
Supplied machine conditions are not restartable.
no_room_for_dsb:
No room available for device status block.
no_room_for_lock:
The record block is too small to contain a lock.
no_s_permission:
Status permission missing on directory containing entry.
no_search_list:
Search list is not in search segment.
no_search_list_default:
Search list has no default.
no_set_btcnt:
Unable to set the bit count on the copy.
no_stmt_delim:
A statement delimiter is missing.
no_table:
The specified table does not exist.
no_term_type:
Unknown terminal type.
no_terminal_quota:
An upgraded directory must have terminal quota.
no_trap_proc:
Cannot find procedure to call link trap procedure.
no_w_permission:
No write permission on entry.
no_wdir:
No working directory set for this process.
no_wired_structure:
No wired structure could be allocated for this device request.
noalloc:
There is no room to make requested allocations.
noarg:
Expected argument missing.
nodescr:
Expected argument descriptor missing.
noentry:
Entry not found.
nolinkag:
No/bad linkage info in the lot for this segment.

nolot:
No linkage offset table in this ring.
nomatch:
Use of star convention resulted in no match.
non_matching_uid:
Unique id of segment does not match unique id argument.
nonamerr:
The operation would leave no names on entry.
nondirseg:
This operation is not allowed for a segment.
nopart:
The partition was not found.
noprtdmp:
No part dump card in config deck.
nostars:
Star convention is not allowed.
not_a_branch:
Entry is not a branch.
not_a_wait_channel:
Event channel is not a wait channel.
not_a_valid_iocb:
The supplied pointer does not point to a valid IOCB.
not_abs_path:
Pathname supplied is not an absolute pathname.
not_act_fnc:
This active function cannot be invoked as a command.
not_archive:
Segment is not an archive.
not_attached:
I/O switch (or device) is not attached.
not_bound:
Segment is not bound.
not_closed:
I/O switch is not closed.
not_detached:
I/O switch is not detached.
not_done:
Not processed.
not_in_trace_table:
Entry not found in trace table.
not_link:
This operation may only be performed on a link entry.
not_open:
I/O switch is not open.
not_privileged:
This operation requires privileged access not given to this process.
not_ring_0:
Signaller called while not in ring 0.
not_seg_type:
Segment not of type specified.
notadir:
Entry is not a directory.
notalloc:
Allocation could not be performed.

nrmkst:
There is no more room in the KST.
null_brackets:
Null bracket set encountered.
null_dir:
The directory specified has no branches.
null_info_ptr:
Pointer to required information is null.
obsolete_function:
Attempt to perform an operation which is obsolete.
odd_no_of_args:
Odd number of arguments.
old_dim:
Old DIM cannot accept new I/O call.
oldnamerr:
Name not found.
oldobj:
Obsolete object segment format.
oob_stack:
User stack space exhausted.
oob_stack_ref:
Attempt to reference beyond end of stack.
oosw:
There was an attempt to reference a directory which is out of service.
order_error:
An error occured while processing the order request.
out_of_bounds:
Reference is outside allowable bounds.
out_of_main_memory:
There is insufficient memory to wire the requested I/O buffer.
out_of_sequence:
A call that must be in a sequence of calls was out of sequence.
out_of_window:
The point or region specified lies outside the window.
outward_call_failed:
Error making outward call after stack history destroyed.
overlapping_more_responses:
The yes and no response characters are not distinct.
pathlong:
Pathname too long.
picture_bad:
The picture contains a syntax error.
picture_scale:
The picture scale factor not in the range -128:+127.
picture_too_big
The normalized picture exceeds 64 characters.
positioned_on_bot:
Tape positioned on leader.
private_volume:
The logical volume is private.
proj_not_found:
Specified project not found.
process_stopped:
Target process in stopped state.

process_unknown:
Target process unknown or in deactivated state.
pv_is_in_lv:
The physical volume is already in the logical volume.
pv_no_scavenge:
The physical volume cannot be scavenged.
pvid_not_found:
The physical volume is not mounted.
quit_term_abort:
Aborted by quit or term.
r0_refname:
Attempt to use reference names in ring 0.
rcp_attr_not_permitted:
Some attribute specified is not permitted for this resource.
rcp_attr_protected:
Some attribute specified is protected.
rcp_bad_attributes:
Resource attribute specification is invalid.
rcp_no_auto_reg:
The resource cannot be automatically registered.
rcp_no_registry:
The registry was not found.
record_busy:
Record locked by another process.
recoverable_error:
Requested operation completed but non-fatal errors or inconsistencies were encountered.
recursion_error:
Infinite recursion.
refname_count_too_big:
The reference name count is greater than the number of reference names.
request_id_ambiguous:
The specified request id matches multiple requests.
request_not_recognized:
Request not recognized.
request_pending:
Processing of request has not been completed.
reservation_failed:
The resource reservation request has failed.
resource_assigned:
Resource already assigned to requesting process.
resource_attached:
Resource already attached to the requesting process.
resource_bad_access:
Resource not accessible to the requesting process.
resource_free:
This operation not allowed for a free resource.
resource_locked:
The resource is locked.
resource_not_free:
The resource is not free.
resource_not_modified:
Specified resource property may not be modified in this manner.

resource_reserved:
The resource is otherwise reserved.
resource_spec_ambiguous:
Resource specification supplied is incomplete.
resource_type_inappropriate:
Resource type is inappropriate for this request.
resource_type_unknown:
Resource type unknown to the system.
resource_unassigned:
Resource not assigned to requesting process.
resource_unavailable:
No appropriate resource available.
resource_unknown:
Resource not known to the system.
retrieval_trap_on:
Retrieval trap on for file special user is trying to access.
root:
The directory is the ROOT.
rqover:
Record quota overflow.
run_unit_not_recursive:
There can be only one run unit at a time.
safety_sw_on:
Attempt to delete segment whose safety switch is on.
salv_pdir_procterm:
Fatal salvaging of process directory.
sameseg:
Attempt to specify the same segment as both old and new.
scavenge_aborted:
The volume scavenge has been terminated abnormally.
scavenge_in_progress:
The volume is being scavenged.
scavenge_process_limit:
Maximum number of simultaneous scavenges exceeded.
seg_busted:
Entry has been damaged. Please type "help damaged_segments.gi".
seg_deleted:
The segment has been deleted.
seg_not_found:
Segment not found.
seg_unknown:
Segment not known to process.
segfault:
Segment fault occurred accessing segment.
segknown:
Segment already known to process.
seglock:
The segment is already locked.
segnamedup:
Name already on entry.
segno_in_use:
The segment number is in use.
short_record:
Record is too short.

signaller_fault:
Fault in signaller by user's process.

size_error:
The size condition has occured.

smallarg:
Argument size too small.

soos_set:
Security-out-of-service has been set on some branches due to AIM inconsistency.

special_channel:
The event channel specified is a special channel.

special_channels_full:
All available special channels have been allocated.

stack_not_active:
The requested ring-0 stack is not active.

stack_overflow:
Not enough room in stack to complete processing.

strings_not_equal:
Strings are not equal.

tape_error:
Tape error.

termination_requested:
Process terminated because of system defined error condition.

time_too_long:
Specified time limit is too long.

timeout:
The operation was not completed within the required time.

too_many_acl_entries:
Access control list exceeds maximum size.

too_many_args:
Maximum number of arguments for this command exceeded.

too_many_buffers:
Too many buffers specified.

too_many_names:
Name list exceeds maximum size.

too_many_read_delimiters:
Too many read delimiters specified.

too_many_refs:
Unable to increment the reference count because of upper bound limit.

too_many_sr:
Too many search rules.

too_many_links:
There are too many links to get to a branch.

too_many_tokens
The date/time string contains more tokens than the routine is prepared to handle.

trace_table_empty:
Trace table is empty.

trace_table_full:
Trace table is full.

translation_aborted:
Fatal error. Translation aborted.

translation_failed:
Translation failed.

typename_not_found:
Typename not found.
unable_to_check_access:
It was not possible to complete access checking - access denied.
unable_to_do_io:
Unable to perform critical I/O.
unbalanced_brackets:
Brackets do not balance.
unbalanced_parentheses:
Parentheses do not balance.
unbalanced_quotes:
Quotes do not balance.
undefined_mode:
Mode not defined.
undefined_order_request:
Undefined order request.
undefined_ptrname:
Unrecognizable ptrname on seek or tell call.
unexpected_condition:
An unexpected condition was signalled during the operation.
unexpected_ft2:
Attempt to execute instruction containing a fault tag 2.
unexpired_file:
Unable to overwrite an unexpired file.
unexpired_volume:
Unable to continue processing on unexpired volume.
unimplemented_ptrname:
Pointer name passed to seek or tell not currently implemented by it.
unimplemented_version:
This procedure does not implement the requested version.
uninitialized_volume:
Unable to continue processing on uninitialized volume.
unknown_tp:
The specified tuning parameter does not exist.
unknown_zone:
The time zone is not acceptable.
unrecognized_char_code:
Volume recorded in unrecognized character code.
unregistered_volume:
The specified detachable volume has not been registered.
user_not_found:
User-name not on access control list for branch.
vol_in_use:
The volume is in use by another process.
volume_busy:
The requested volume is not available.
volume_not_loaded:
The requested volume is not loaded.
volume_type_unknown:
Volume type unknown to the system.
vtoc_io_err:
Unrecoverable data-transmission error on VTOC.
vtoce_connection_fail:
Some directory or segment in the pathname is not listed in the VTOC.

vtoce_free:
The VTOCE is already free.
wakeup_denied:
Insufficient access to send wakeup.
wrong_channel_ring:
An event channel is being used in an incorrect ring.
wrong_no_of_args:
Wrong number of arguments supplied.
zero_length_seg:
Zero length segment.

CONDITIONS

Status codes enable a calling procedure to take action on an unusual occurrence only after the procedure encountering the occurrence has returned. It is sometimes necessary for a calling procedure to gain control immediately upon encountering an unusual occurrence, so that it can decide what action to take. If the calling procedure decides to take corrective action, it can then continue execution from the point of the occurrence. Unusual occurrences can also be detected at times when no error is expected, so that execution of the program must be interrupted immediately. This is the purpose of the Multics condition mechanism (described in "Multics Condition Mechanism" below). Not all conditions can be corrected to the point where execution of the program can be continued. See description of system conditions below for information about specific conditions.

The condition mechanism is also used for error reporting in cases where the errors a procedure can detect occur too infrequently and speed is too important to have a status code argument.

The Multics system invokes the condition mechanism upon encountering certain unusual occurrences during the execution of a program. The Multics standard user environment acts upon these system-generated occurrences, as well as occurrences generated by user programs if the user programs do not do so themselves. A list of occurrences that cause the system to invoke the condition mechanism, and the action taken by the Multics standard user environment if it is invoked to act upon these occurrences, is given in "List of System Conditions and Default Handler" below. Methods of signalling conditions from user programs are discussed in "Signalling Conditions in a User Program" below.

Multics Condition Mechanism

The condition mechanism is a facility of the Multics system that notifies a program of an exceptional condition detected during its execution. A condition is a state of the executing process. Each condition that is detected is identified by a condition name. For example, division by zero is a condition identified by the condition name, zerodivide.

A condition can be detected by the system or by a user program. When a condition is detected, it is signalled. A signal causes a block activation of the most recently established on unit for the condition. Thus, by establishing an on unit, a program arranges with the system to receive control when conditions of interest to it are detected and signalled.

An on unit can be a begin block or independent statement, or it can be a procedure entry. A program (an activation of a procedure block or begin block) can establish a begin block or an independent statement as an on unit for a particular condition by executing a PL/I on statement that names that condition.

When an on unit is activated, it can take any action to handle a condition. Typically, the on unit might try to rectify the circumstances that caused the condition and then restart execution of the interrupted program at the point where the condition was detected; or it might abort execution of the program by performing a nonlocal transfer to a location within the interrupted program or to one of its callers.

All of the on units established by a block activation are reverted when that block activation terminates by returning to its caller or when it is aborted by a nonlocal transfer. An on unit for a particular condition can be explicitly reverted by executing a PL/I revert statement or by executing another on statement that names the condition. Therefore, each block activation can have no more than one on unit established for each condition at any given time; however, there can be as many on units established for a particular condition as there are block activations. Signalling a condition causes a block activation of the most recently established on unit for that condition. Normally, this is the only on unit that is activated, even though other on units for the condition were established by preceding block activations.

The effect of this scheme is that, once a block activation has established an on unit for a condition, any occurrence of the condition activates that on unit. This remains true only until the block activation is terminated or until the on unit is reverted and as long as no descendant block activation establishes an on unit for the condition.

Generally, procedures that can take action when a condition is detected should establish an on unit for that condition. Of those block activations that have established an on unit for the condition, the most recently established on unit is activated.

Example of the Condition Mechanism

The example below is presented to illustrate the mechanism discussed above. It is not meant to illustrate typical or recommended use of the condition mechanism.

```
Example:  proc;

          declare Sub1 external entry;
          declare Sub2 external entry;
          declare c fixed bin;
          declare wrong_way condition;

          on wrong_way begin;                          (1)
            .
            .
            .
          end;

          call Sub1;                                   (2)

          c = 2;                                       (3)

          call Sub2;                                   (4)

          end Example;

Sub1:     proc;

          declare a fixed bin;
          declare wrong_way condition;

          a = 0;                                       (S1)

          on wrong_way begin;                          (S2)
            .
            .
            .
          end;

          a = 1;                                       (S3)

          end Sub1;
```

```
Sub2:     proc;

          declare b fixed bin;
          declare wrong_way condition;

          b = 1;                                        (S4)

          on wrong_way begin;                           (S5)
            .
            .
            .
          end;

          b = 2;                                        (S6)

          revert wrong_way;                             (S7)

          b = 3;                                        (S8)

          end Sub2;
```

In the above example, if procedure Example is called, the executable statements are executed in the order (1), (2), (S1), (S2), (S3), (3), (4), (S4), (S5), (S6), (S7), (S8) under normal circumstances. However, if the wrong_way condition is detected and signalled during the execution of (S1), then the on unit established for the wrong_way condition by Example is activated because Sub1 has not established an on unit for the wrong_way condition at this time. If the on unit simply corrects the circumstances that caused the wrong_way condition and returns, then execution resumes in (S1) from the point of interruption. If the wrong_way condition is detected and signalled during the execution of statement (S3), then the on unit established in Sub1 is activated because Sub1 has established the most recent on unit for this condition. If the wrong_way condition is signalled during (3), the on unit established by Example is activated because the block activation for Sub1 has been terminated and its on unit is no longer established. If the wrong_way condition is signalled during (S8), the on unit established in Example is activated because Sub2 explicitly reverted the on unit it had previously established, making Example's on unit the most recently established on unit for the wrong_way condition.

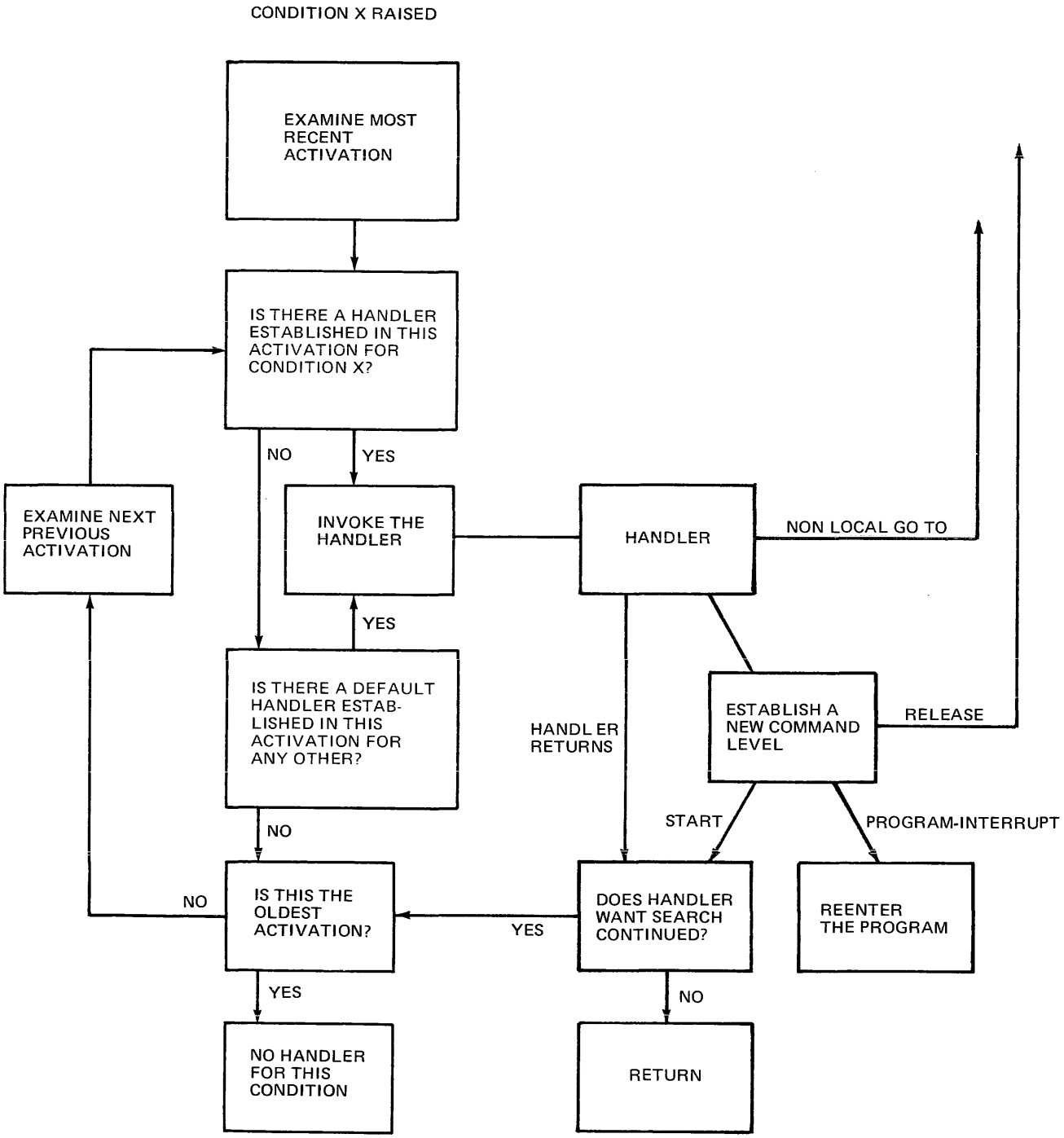
On Unit Activated by All Conditions

The above description indicates how on units can be established for specific conditions. It is sometimes desirable to handle any and all conditions that occur. To do this, a block activation can establish an on unit for the any_other condition. When a particular condition is signalled, the any_other on unit established by the block activation is activated if no specific on unit for the condition was established by the block activation, and if no on unit for that condition or the any_other condition was established by a more recent block activation. In other words, when a condition is signalled, each block activation, starting with the most recent, is inspected for an on unit established for that specific condition and, if none is found, each block is inspected for an established any_other on unit. The first such specific or any_other on unit found is the one that is activated. As is the case with on units for specific conditions, only one any_other on unit can be established by a given block activation. Establishing a second any_other on unit simply overwrites the first.

Continuation of Search

If an on unit cannot adequately handle the condition, it causes the condition mechanism to continue to search for another on unit by calling the continue_to_signal_ subroutine (described in the Subroutines manual) and then returning.

As a summary, the flow diagram of Figure 7-1 illustrates the algorithm used by the condition mechanism to determine which on unit to activate when a condition is signalled. The action taken when no on unit can be found for a condition is described in "Interaction with the Multics Ring Structure" below.


CONDITION X RAISED
EXAMINE MOST RECENT ACTIVATION
IS THERE A HANDLER ESTABLISHED IN THIS ACTIVATION FOR CONDITION X?
NO
YES
EXAMINE NEXT PREVIOUS ACTIVATION
INVOKE THE HANDLER
HANDLER
NON LOCAL GO TO
YES
IS THERE A DEFAULT HANDLER ESTABLISHED IN THIS ACTIVATION FOR ANY OTHER?
HANDLER RETURNS
ESTABLISH A NEW COMMAND LEVEL
RELEASE
START
PROGRAM-INTERRUPT
NO
NO
IS THIS THE OLDEST ACTIVATION?
YES
DOES HANDLER WANT SEARCH CONTINUED?
REENTER THE PROGRAM
YES
NO
NO HANDLER FOR THIS CONDITION
RETURN

Interaction with the Multics Ring Structure

The condition mechanism interacts with the Multics ring structure. The above description of how an on unit is selected for activation applies only to block activations within a single ring. When a condition is signalled in a particular ring, the algorithm of Figure 7-1 is followed for the block activations in that ring. If no on unit for the condition is found in that ring, then the ring is abandoned and the same condition is signalled in the higher ring that called the abandoned ring. This process is repeated until all existing rings have been abandoned, indicating that this process has not established an on unit for the condition being signalled, in which case the process is terminated. For more information, see "Action Taken by the Default Handler" below.

Nonstandard Location of On Unit for Special Conditions

The standard searching mechanism is bypassed for certain conditions. Before looking in the stack for on units, a special table is examined to determine if a "static handler" has been established for the condition. If so, the static handler is invoked and has the option of passing the condition on to the normal mechanism. Static handlers are used by the system to handle certain system events that users should not attempt to handle. Consult static_handlers.incl.pl1 for the conditions that may have static handlers.

Action Taken by the Default Handler

Some conditions are routinely handled by the system's default on unit (in the absence of a user-supplied on unit) by printing a message on the user's terminal to alert him that the condition has occurred and his process has returned to command level. These conditions are denoted in "List of System Conditions and Default Handler" below by the following: "Default action: prints a message and returns to command level."

In many cases, the subroutine that is executing when a condition is detected is a system or PL/I support subroutine that is of little interest to the user. In such cases, the user needs to know the location at which the most recent nonsupport subroutine was executing before the condition was detected. To accomplish this, the default on unit hunts through the block activations that precede the support subroutine until it finds the first nonsupport subroutine; it then indicates that the condition was detected while executing at a location within that nonsupport subroutine.

System Condition Wall

The system sets up a handler for the any_other condition when the user ring environment is established; this handler serves to establish the system's default handler (see "Action taken by Default Handler" above). The system similarly sets up a handler for the any_other condition each time the system's default handler returns to command level. This handler serves to establish the system default handler as the handler for all conditions raised by programs invoked at the new command level, effectively replacing the handlers of the program which caused the default handler to be invoked. Thus, a "condition wall" is set up between programs raising conditions which have no handlers for them and programs run at a new command level thereafter. Invocations of command level from programs (via calls to cu_$cl) or explicit invocation of the default_error_handler_ subroutine do not cause this wall to be set up. Any program that wishes to set up an explicit condition wall can set up a handler for the any_other condition. The statement:

```
on any_other system;
```

will cause a condition wall which invokes the standard system default handler in all cases.

The condition wall is transparent to the program_interrupt and finish conditions; this allows the program_interrupt condition to be used to reenter suspended environments.

Signalling Conditions in a User Program

A user program can signal a condition by executing a PL/I signal statement that names that condition. If descriptive arguments are to be passed to the on unit, the signal_ subroutine (described in the Subroutines manual) should be called with the condition name as an argument. If the on unit activated by the signal returns, the user program should retry the operation that was interrupted by the condition.

Obtaining Additional Information About a Condition

An on unit usually needs information about the circumstances under which it was activated. The find_condition_info_ subroutine (described in the Subroutines manual) makes such information available to an on unit. The information might include machine conditions (i.e., the processor state) or other information describing the condition in question. The information that is available when system-detected conditions are signalled is listed in "Machine Condition Data Structure" and "Information Header Format" below.

Machine Condition Data Structure

As discussed above, information is available that describes the state of the processor at the time a hardware condition (fault) was raised. This data structure is declared in the include file mc.incl.pl1 in the following form:

```
dcl 1 mc based (mc_ptr)        aligned,
     2 prs (0:7)                  ptr,
     (2 regs,
        3 x (0:7)                 bit(18),
        3 a                       bit(36),
        3 q                       bit(36),
        3 e                       bit(8),
        3 reserved                bit(64),
      2 scu (0:7)                 bit(36),

      2 errcode                   fixed bin(35),
      2 reserved2                 bit(72),
      2 fault_time                fixed bin(71),
      2 reserved3 (0:7)           bit(36)) unaligned;
```

STRUCTURE ELEMENTS

prs
is the contents of the eight pointer registers at the time the condition occurred.

regs
is the contents of the other registers at the time the condition occurred.

x
is the contents of the eight index registers.

a
is the a register contents.

q
is the q register contents.

e
is the exponent register contents.

scu
is the stored control unit, expanded below.

errcode
is the fault error code. Refer to "List of System Status Codes and Meanings" earlier in this section.

fault_time
is the time the condition occurred.

NOTE: In the above declaration and in the declarations that follow, "reserved" is reserved for use by the system.

The stored control unit is declared as follows:

```
dcl 1 scu                aligned,
        (2 ppr,
          3 prr          bit(3),
          3 psr          bit(15),
        2 reserved4      bit(18),
        2 reserved5      bit(30),
        2 fi_num         bit(5),
        2 fi_flag        bit(1),
        2 tpr,
          3 trr          bit(3),
          3 tsr          bit(15),
        2 reserved6      bit(18),
        2 reserved7      bit(30),
        2 tpr_tbr        bit(6),
        2 ilc            bit(18),
        2 ir,
          3 zero         bit(1),
          3 neg          bit(1),
          3 carry        bit(1),
          3 ovfl         bit(1),
          3 eovf         bit(1),
          3 eufl         bit(1),
          3 oflm         bit(1),
          3 tro          bit(1),
          3 par          bit(1),
          3 parm         bit(1),
          3 bm           bit(1),
          3 tru          bit(1),
          3 mif          bit(1),
          3 abs          bit(1),
          3 reserved     bit(4),
        2 ca             bit(18),
        2 reserved8      bit(18),
        2 even_inst      bit(36),
        2 odd_inst       bit(36)) aligned;
```

STRUCTURE ELEMENTS

ppr
: is the procedure pointer register contents.

  prr
  : is the ring number portion of ppr.

  psr
  : is the segment number portion of ppr.

fi_num
: is the fault/interrupt number.

fi_flag
is the fault/interrupt flag.
"0"b interrupt
"1"b fault

tpr
is the temporary pointer register contents.

trr
is the ring number portion of tpr.

tsr
is the segment number portion of tpr.

tpr_tbr
is the bit offset portion of tpr.

ilc
is the instruction counter contents.

ir
is the contents of indicator registers.

zero
zero indicator.

neg
negative indicator.

carry
carry indicator.

ovfl
overflow indicator.

eovf
exponent overflow.

eufl
exponent underflow.

oflm
overflow mask.

tro
tally runout.

par
parity error.

parm
parity mask.

bm
not bar mode.

tru
truncation mode.

mif
mid instruction fetch.

abs
absolute mode.

ca
is the computed address.

even_inst
the instruction causing the fault is stored here.

odd_inst
the next sequential instruction is stored here if ilc (see above) is even.

Information Header Format

A standard header is required at the beginning of each information structure provided to an on unit. Except for the header, this info is particular to the condition in question and varies among conditions. The condition info structure should be constructed using condition_info_header.incl.pl1 and the PL/1 "like" feature--see the PL/1 condition structure below for an example. The format of the header is:

```
dcl 1 condition_info_header    aligned based (condition_info_header_ptr),
      2 length                 fixed bin,
      2 version                fixed bin,
      2 action_flags           aligned,
        3 cant_restart         bit(1) unaligned,
        3 default_restart      bit(1) unaligned,
        3 quiet_restart        bit(1) unaligned,
        3 support_signal       bit(1) unaligned,
        3 pad                  bit(32) unaligned,
      2 info_string            char(256) var,
      2 status_code            fixed bin(35);
```

STRUCTURE ELEMENTS

length
is the length of the structure in words.

version
is the version number of the structure of the particular condition that was raised. It is documented along with the rest of the info_structure for each condition in the pages that follow.

action_flags
indicate appropriate behavior for a handler.

cant_restart
indicates that a handler should never attempt to return to the signalling procedure.

default_restart
resumes computation with no further action on the handler's part except printing the message in info_string and returning.

quiet_restart
resumes computation with no further action except a return.

support_signal
indicates that the error is being signaled on behalf of another procedure. Any error messages produced by the signal will refer to the caller of the procedure that signaled the condition, rather than to the procedure that signalled the condition.

pad
is reserved for future expansion and must be "0"b.

info_string
is a printable message about the condition.

status_code
if nonzero, is a code interpretable by the com_err_ subroutine that further defines the condition.

If no action flag is set, restarting is possible, but its success depends on the action taken by the handler.

PL/I Condition Data Structure

Most of the PL/I conditions have the data structure described below. Only the items associated with a particular instance of a condition are filled in. The relevant information should be obtained from the PL/I defined ondata structure (beyond the header) since it is primarily an implementation vehicle for the PL/I condition built-in functions.

For brevity, the data structure item of PL/I conditions that use this data structure is listed as "the standard PL/I data structure." See pl1_info.incl.pl1.

```
dcl 1 pll_info                  aligned based (pll_info_ptr),
        2 header                 aligned like condition_info_header,
        2 id                     char(8) init ("pliocond"),
        2 content_flags          aligned,
          (3 v1_sw,
           3 oncode_sw,
           3 onfile_sw,
           3 file_ptr_sw,
           3 onsource_sw,
           3 onchar_sw,
           3 onkey_sw,
           3 onfield_sw)         bit(1) unaligned,
        2 oncode                 fixed bin(35),
        2 onfile                 char(32) aligned,
        2 file_ptr               ptr,
        2 onsource               char(256) var,
        2 oncharindex            fixed bin,
        2 onkey_onfield          char(256) var;
```

STRUCTURE ELEMENTS

header
    is the same as the information in the header format above. (The version field of the header is condition one.)

id
    identifies this structure as belonging to a PL/I condition.

v1_sw
    indicates that the condition was raised by a version 1 PL/I procedure.
    "1"b condition was raised by version 1
    "0"b condition was not raised by version 1

oncode_sw
    indicates that the structure contains a valid oncode.
    "1"b oncode valid
    "0"b no valid oncode present

onfile_sw
    indicates that a file name has been copied into the structure.
    "1"b name copied
    "0"b name is not copied

file_ptr_sw
    indicates that there is a file associated with this condition.
    "1"b file associated
    "0"b file is not associated

onsource_sw
    indicates that there is a valid onsource string for this condition.
    "1"b valid onsource string
    "0"b no valid onsource string present

onchar_sw
indicates that there is a valid onchar index in this structure.
"1"b valid onchar index
"0"b no valid onchar index present

onkey_sw
indicates that there is a valid onkey string in this structure.
"1"b valid onkey string
"0"b no valid onkey string present

onfield_sw
indicates that there is a valid onfield string in this structure.
"1"b valid onfield string
"0"b no valid onfield string present

oncode
is the condition's oncode if oncode_sw is equal to "1"b.

onfile
is the onfile string if onfile_sw is equal to "1"b.

file_ptr
is a pointer to a file value if file_ptr_sw is equal to "1"b.

onsource
is the onsource string if onsource_sw is equal to "1"b.

oncharindex
is the character offset in onsource of the erroneous character if onchar_sw is equal to "1"b.

onkey_onfield
is the onkey string if onkey_sw is equal to "1"b and is the onfield string if onfield_sw is equal to "1"b.

SYSTEM CONDITIONS AND DEFAULT HANDLER

System conditions are signalled to report certain unusual occurrences. The signalling and handling of conditions in general is described in "Multics Condition Mechanism" above. The following discussion lists the conditions signalled by system procedures and the default actions taken for each. The default on unit is invoked if no other user or system on unit has been established for the condition. The conditions are listed in alphabetical order by name.

When present, the parenthetical type designator at the right margin on the same line with the name indicates that the condition is either:

1. defined by the PL/I language; or

2. due to a hardware fault or an error encountered while processing a hardware fault (indicating that a processor state description is available).

Otherwise, the condition is neither of these.

Five items follow each condition name:

Cause: is the reason the condition is signalled;

Default action: is a brief description of the action taken by the default on unit;

Restrictions: indicates when the user should not attempt to handle the condition and when the system handles the condition before any search for user on units;

Restartability: indicates what actions, if any, are necessary to restart after an occurrence of the condition. Restarting is accomplished either by returning from an on unit or by issuing the start command from command level. Restartability has four classifications:

1. immediately restartable: execution continues (default action) if an on unit returns without doing anything; this default action is usually documented.

2. conditionally restartable: restarting is meaningful only if some obvious corrective action (such as resetting access) is taken.

3. conditionally restartable by modifying machine conditions: restarting is meaningful only if certain items in the saved machine state are modified. This is a sophisticated process that is discussed in the *Multics Processor Manual*, Order No. AL39.

4. not restartable: restarting produces undefined and usually erroneous results.

No condition that originally occurred in a lower ring is restartable since the stack history at the time of occurrence is discarded when a higher ring is entered in the search for a handler.

The decision to restart need not be limited to the on unit itself. In many cases, the on unit returns to command level with the stack history preserved so the user can decide whether to take corrective action and whether restarting is appropriate.

Data Structure: is the PL/I declaration of the data that can be pointed to by info_ptr, the fourth argument available to a condition handler. Unless otherwise specified, it is not generally useful for the handler to change the values of variables in the data structure.

List of System Conditions

In the list of system conditions that follow, one default action description occurs frequently. For brevity, it is listed as:

"prints a message and returns to command level"

to mean:

> "an error message is printed on the error_output switch, and the user is placed at command level with a higher level stack frame than before the condition was signalled."

When a user receives this message, his stack is intact and the history of the error is preserved. The user can retain the stack for further debugging activities or he can release it. (See the description of the release and start commands in the Commands manual.)

active_function_error

Cause: the user incorrectly used an active function in a command line. The active_fnc_err_ subroutine signals this condition if a command is invoked as an active function; the com_err_ subroutine also may signal this condition if a command is invoked as an active function (see the Subroutines manual).

Default action: prints a message and returns to command level.x)

Restrictions: none.

Restartability: restartable.

Data structure:

```
dcl 1 com_af_error_info   aligned based,
      2 header            aligned like condition_info_header,
      2 name_ptr          ptr,
      2 name_lth          fixed bin,
      2 errmsg_ptr        ptr,
      2 errmsg_lth        fixed bin,
      2 max_errmsg_lth    fixed bin,
      2 print_sw          bit(1);
```

Structure elements:

header
: is the same as in the information format header above, except that the version number is 3. See com_af_error_info.incl.pl1.

name_ptr
: is a pointer to a character string containing the name of the procedure that called the active_fnc_err_ subroutine.

name_lth
is the length of the name of the procedure that called the active_fnc_err_ subroutine.

errmsg_ptr
is a pointer to a character string containing the error message prepared by the active_fnc_err_ subroutine. A handler might wish to alter that message.

errmsg_lth
is the significant length of the error message prepared by the active_fnc_err_ subroutine. This datum can be changed by the handler.

max_errmsg_lth
is the size of the character string containing the error message prepared by the active_fnc_err_ subroutine.

print_sw
indicates whether the error message is printed by the active_fnc_err_ subroutine if and when the handler returns control to it. This datum can be changed by the handler.
"1"b message is printed
"0"b message is not printed

alrm (hardware)

Cause:	a real-time alarm occurred a specified length of time after a call by the use to the timer_manager_$alarm_call entry point (to set the alarm). See the description of the timer_manager_ subroutine in the Subroutines manual.
Default action:	the handler looks up the alarm that is expected at the time this one occurred, and calls the appropriate user-specified procedure. When (if) this procedure returns, the user's process is returned to the point at which it was interrupted.
Restrictions:	the user should not attempt to handle this condition.
Restartability:	immediately restartable.
Data structure:	none.
Note:	this condition is normally handled by a static handler and, therefore, the stack is never searched.

area (PL/I)

Cause: the user attempted either to allocate storage in an area that had insufficient space remaining to generate the storage needed, or to assign one area to another, and the second had insufficient space to hold the storage allocated in the first.

Default action: prints a message on the error_output switch and signals the error condition. Upon a normal return, the attempted allocation is retried in case the user has freed some storage from an area in the interim.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

bad_area_format

Cause: a block of words could not be allocated in an area because the area was damaged. The damage was either to certain words in the area header that define the type of area or to the threads connecting blocks of free storage.

Default action: prints a message and returns to command level.

Restriction: none.

Restartability: sometimes conditionally restartable by fixing the area.

Data structure: none.

*bad_dir_*

Cause: the supervisor has detected damage to a directory which cannot be repaired by the automatic recovery procedures.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

bad_outward_call (hardware)

Cause: the user attempted to make an invalid call to an outer ring.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

cleanup

Cause: a program activation is being aborted as a result of a nonlocal transfer to a location in an earlier procedure activation. Because it is not going to return normally, the program activation may want to regain control briefly to reset static variables, etc. This is not an ordinary condition because every cleanup on unit established between the frame of the program activation performing the transfer and the frame of the target program activation is invoked.

Default action: none.

Restrictions: an on unit for cleanup or any of its dynamic decendants must not do a nonlocal goto since this would interfere with the one already in progress. The user should not signal this condition directly.

Restartability: immediately restartable. The on unit must return.

Data structure: none.

*command_abort_*

Cause: a FORTRAN or PL/I program executed a stop statement, or the user invoked the stop_run command; this is signalled only when the user is not in a run unit.

Default action: transfers to the activation of the command processor that invoked the program, unwinding the stack; the next command, if any, on the command line is then processed. If a subsystem command processor does not have a handler for this condition, the default action is to print a message and return to command level.

Restrictions: user programs other than command processors must not attemqpt to handle this condition.

Restartability: immediately restartable; however, if the condition was signalled from a stop statement, execution continues with the statement following the stop statement. A release is necessary to achieve the desired effect.

Data structure: none.

command_error

Cause: the user incorrectly used a command (such as giving it bad arguments), or a command encountered a situation that prevented it from completing its operation normally. The com_err_ subroutine (described in the Subroutines manual) signals this condition.

Default action: returns to the com_err_ subroutine, which then prints a formatted message on the error output switch. Other more sophisticated handlers could reformat the error message to the individual user's taste, or take some special action depending on the particular condition in question.

Restrictions: none.

Restartability: immediately restartable.

Data structure:

```
dcl 1 com_af_error_info      aligned based,
      2 header               aligned like condition_info_header,
      2 name_ptr             ptr,
      2 name_lth             fixed bin,
      2 errmess_ptr          ptr,
      2 errmess_lth          fixed bin,
      2 max_errmess_lth      fixed bin init(256),
      2 print_sw             bit(1) init("1"b);
```

Structure elements:

header
is the same as in the information header format above, except that the version number is 3. See com_af_error_info.incl.pl1.

name_ptr
is a pointer to a character string containing the name of the procedure that called the com_err_ subroutine.

name_lth
is the length of the name of the procedure that called the com_err_ subroutine.

errmess_ptr
is a pointer to a character string containing the error message prepared by the com_err_ subroutine. A handler might wish to alter that message.

errmess_lth
is the significant length of the error message prepared by the com_err_ subroutine. This datum can be changed by the handler.

max_errmess_lth
is the size of the character string containing the error message prepared by the the com_err_ subroutine.

print_sw
indicates whether the error message is printed by the com_err_ subroutine. This datum can be set by the handler.
"1"b message is printed
"0"b message is not printed

command_query_error

Cause: the user specified a handler for the command_question condition that did not return a yes or no answer when the data structure element indicated that a yes or no answer was required. The command_query_ subroutine (described in the Subroutines manual) signals this condition.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

Data structure: none.

command_question

Cause: a command is asking a question of the user. The command_query_ subroutine signals this condition.

Default action: returns to the command_query_ subroutine, which then prints the question on the user_output switch. Other more sophisticated handlers could supply a preset answer, modify the question, or suppress its printing. See the data structure below for details.

Restrictions: none.

Restartability: immediately restartable.

Data structure:

```
dcl 1 command_question_info   aligned based,
      2 header                aligned like condition_info_header
      2 query_code            fixed bin(35),
      2 question_sw           bit(1) init ("1"b) unaligned,
      2 yes_or_no_sw          bit(1) unaligned,
      2 preset_sw             bit(1) init("0"b) unaligned,
      2 answer_sw             bit(1) init("1"b) unaligned,
      2 name_ptr              ptr,
      2 name_lth              fixed bin,
      2 question_ptr          ptr,
      2 question_lth          fixed bin,
      2 max_question_lth      fixed bin,
      2 answer_ptr            ptr,
      2 answer_lth            fixed bin,
      2 max_answer_lth        fixed bin,
      2 question_iocbp        ptr,
      2 answer_iocbp          ptr,
      2 repeat_time           fixed bin(71);
```

Structure elements:

header
: is the same as in the information header format above, except that the version number is 6. See command_question_info.incl.pl1.

query_code
: is additional qualifying information passed by the caller of command_query_.

question_sw
: indicates whether the command_query_ subroutine should print the question. This datum can be set by the handler.
"1"b question is printed
"0"b question is not printed

yes_or_no_sw
indicates whether the command_query_ subroutine expects the preset answer (if any) returned by the handler to be either yes or no. In this case, if the handler returns any other string, the command_query_ subroutine signals the the command_query_error condition.
"1"b answer either yes or no
"0"b any answer accepted

preset_sw
indicates whether the handler is returning in the character string pointed to by answer_ptr a preset answer to the command_query_ subroutine. In that case, the command_query_ subroutine returns the preset answer to its caller. That is, it does not attempt to obtain an interactive response by reading from the user_input switch. Leading and trailing blanks and the terminal newline character (if present) are removed. This datum can be changed by the handler.
"1"b preset answer returned
"0"b no answer returned

answer_sw
indicates whether the command_query_ subroutine should print the preset answer (if any). This datum can be changed by the handler.
"1"b print answer
"0"b no answer printed

name_ptr
is a pointer to a character string containing the name of the procedure that called the command_query_ subroutine.

name_lth
is the length of the name of the procedure that called the command_query_ subroutine.

question_ptr
is a pointer to a character string containing the question prepared by the command_query_ subroutine. A handler might wish to alter that question.

question_lth
is the significant length of the question pointed to by question_ptr. This datum can be changed by the handler.

max_question_lth
is the size of the character string pointed to by question_ptr.

answer_ptr
is a pointer to a character string that can be used by the handler to return a preset answer.

answer_lth
is the significant length of the preset answer pointed to by answer_ptr. This datum can be changed by the handler.

max_answer_lth
is the size of the character string pointed to by answer_ptr.

question_iocbp
is the I/O control block pointer that the command_query_ subroutine uses to write the question if a preset answer is not returned by the handler. The handler may redefine this iocbp.

answer_iocbp
is the I/O control block pointer that the command_query_ subroutine uses to read the answer to the question if a preset answer is not returned by the handler. The handler may redefine this iocbp.

repeat_time
is the number of seconds that the command_query_ subroutine waits, after asking the question, before repeating the question if no answer was given. A value less than 30 indicates that the question is not to be repeated. This datum may be changed by the handler.

conversion (PL/I)

Cause: a PL/I conversion or runtime I/O routine attempted an invalid conversion from character string representation to some other representation. Possible invalid conversions are a character other than 0 or 1 being converted to bit string, and nonnumeric characters where only numeric characters are permitted in a conversion to arithmetic data.

Default action: prints a message on the error_output switch and signals the error condition. Upon a normal return, the conversion is attempted again, using the value of the PL/I onsource pseudovariable as the input character string.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: the standard PL/I data structure.

Note: the user can establish a handler that uses the onchar and onsource builtin functions to alter the invalid character string.

cput (hardware)

Cause: a CPU time interrupt occurred after a user-specified amount of CPU time had passed following a call to the timer_manager_$cpu_call entry point. (See the description of the timer_manager_ subroutine in the Subroutines manual.

Default action: the handler looks up the CPU time interrupt that is expected at this time and calls the appropriate user-specified procedure. When (if) this procedure returns, the process is returned to the point at which it was interrupted.

Restrictions: the user should not attempt to handle this condition.

Restartability: immediately restartable.

Data structure: none.

Note: this condition is normally handled by a static handler and, therefore, the stack is never searched.

create_ips_mask_err

Cause:	the create_ips_mask_ subroutine (described in the Subroutines manual) was called with a name that was not a valid ips signal name.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	not restartable.
Data structure:	none.

cross_ring_transfer (hardware)

Cause:	the user attempted to cross ring boundaries using a transfer instruction. A CALL or RTCD instruction must be used to cross ring boundaries.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	conditionally restartable.
Data structure:	none.

derail (hardware)

Cause:	the user attempted to execute a DRL instruction on the processor.
Default action:	prints a message and returns to command level.
Restrictions:	usually none. However, some subsystems use it for special purposes. When operating within such subsystems, the user should not attempt to handle the condition.
Restartability:	conditionally restartable by modifying machine conditions.
Data structure:	none.

*dm_not_available_*

Cause: the process has tried to use the Data Management System (DMS), but no active DMS exists.

Default action: the DM first reference trap is reset and a message is sent to the error_output I/O switch indicating the DMS is not available for use. The process is left at a new command level.

Restrictions: the user process should not handle this condition unless it will warn the user that DM usage will fail. For example, DM specific commands like transaction and before_journal_status report the unavailability via com_err_.

Restartability: not restartable. The user may try to use Data Management later in the process, however.

Data structure: the standard PL/I data structure.

Note: this can occur if the DM Daemon has not begun or finished booting the DMS, the DMS bootload has been shutdown, or there is no room in the DMS per-system tables to record the necessary information about the process trying to use DMS. The condition is signalled by the first reference trap for the inner ring DM software which does DM per-process initialization.

*dm_shutdown_scheduled_*

Cause: the process is being notified that the scheduled shutdown time for the data management system that the process is using has been set or changed.

Default action: the handler sets two timers. The first one will be set to cause a dm_shutdown_warning_ condition to be signalled prior to the beginning of data management shutdown. The second timer will be set to cause a dm_user_shutdown_ condition to be signalled within the user's process.

Restrictions: none.

Restartability: immediately restartable.

Data structure: none.

Cause: the process has signalled the dm_shutdown_warning_ condition. This is normally caused if the process has handled the dm_shutdown_scheduled_ IPS interrupt from a data management daemon or if the DM system (DMS) being used had a shutdown time scheduled when the process started using the DMS. In the normal case, the process will still be able to use the DMS for a short time before DMS user shutdown will occur (see the dm_user_shutdown_ condition).

Default action: if the process is interactive, a message is printed on the user_i/o I/O switch indicating the date and time the DMS will shut down. No default action is taken for absentee processes; however, any process type may handle the condition. After the default action is taken, the process will continue executing at the point it was interrupted.

Restrictions: the user process should not handle this condition unless it will print some warning to the user if the process is interactive.

Restartability: immediately restartable.

Data structure:

```
dcl 1 dm_shutdown_warning_info      aligned based,
      2 header                      like condition_info_header,
      2 begin_shutdown_time         fixed bin (71),
      2 user_shutdown_time          fixed bin (71),
      2 flags,
        3 dont_print_warning        bit (1) unaligned,
        3 mbz                       bit (35) unaligned,
      2 reason                      char (64);
```

Structure elements:

header
: is the same as in the information header format above. The version number is 1, and info_string is set to the default warning message.

begin_shutdown_time
: is a standard system clock value when the DMS will not longer allow transactions to begin.

user_shutdown_time
is a standard system clock value when user processes are to start DMS shutdown.

flags.dont_print_warning
is a flag that a user handler may turn on to prevent the standard warning message from being output.

flags.mbz
must be set to "0"b.

reason
is the reason the DMS is being shut down.

*dm_user_shutdown_*

Cause: the process has signalled the dm_user_shutdown_ condition. This is normally caused if the process has handled the dm_shutdown_scheduled_ IPS interrupt from a data management daemon or if the DM system (DMS) being used had a shutdown time scheduled when the process started using the DMS. The process has a small amount of time to adjust any non-DMS values associated with DMS use (e.g. a MRDS control.db segment).

Default action: the handler signals this condition to the user process and will call transaction_manager_$user_shutdown after return from any user handlers for the condition. After return from the user_shutdown call, a message will be output on the user_i/o I/O switch that the user's references to DMS have been invalidated and the transaction in progress (if one exists) has been aborted. After the default action is taken, the process will continue executing at the point it was interrupted.

Restrictions: any user handler for the condition signalled should call continue_to_signal_ and then return after doing any work it wishes. If the default action of the static handler is bypassed, the user's condition handler for dm_user_shutdown_ should call transaction_manager_$user_shutdown. Failure to do so will cause the DMS daemon to bump the user at a later time.

Restartability: immediately restartable.

Data structure:

```
dcl 1 dm_user_shutdown_info          aligned based,
      2 header                       like condition_info_header,
      2 flags,
        3 dont_print_warning         bit (1) unaligned,
        3 dont_do_user_shutdown      bit (1) unaligned,
        3 mbz                        bit (34) unaligned,
      2 reason                       char (64);
```

Structure elements:

header
: is the same as in the information header format above. The version number is 1, and info_string is set to the default warning message.

flags.dont_do_user_shutdown
: is a flag a user handler may turn on to prevent the normal call to transaction_manager_$user_shutdown if the process returns to the signalling procedure.

flags.dont_print_warning
: is a flag a user handler may turn on to prevent the standard warning message that his process has done DMS shutdown. This flag is ignored if flags.dont_do_user_shutdown has been turned on.

flags.mbz
: must be set to "0"b.

reason
: is the reason the DMS is being shut down.

endfile (f) (PL/I)

Cause:	a PL/I get or read statement attempted to read past the end of data on the file f.
Default action:	prints a message on the error_output switch and signals the error condition. Upon return from any handler, control passes to the PL/I statement following the statement in which the condition was raised.
Restrictions:	none.
Restartability:	immediately restartable.
Data structure:	the standard PL/I data structure.

endpage (f) (PL/I)

Cause: PL/I inserted the last newline character of the current page into the output stream of file f.

Default action: begins the next page on the file f and returns.

Restrictions: none.

Restartability: immediately restartable. The handler can begin a new page via a PL/I statement of the form:

```
put file (f) page ...  (...  "title"...)
...;
```

or can simply return, permitting the number of lines on the current page to exceed the number normally occurring.

Data structure: the standard PL/I data structure.

error (PL/I)

Cause: some other (more specific) PL/I condition occurred, and its handler signalled the error condition. Alternatively, some PL/I runtime subroutine (e.g., one in the mathematical library) encountered one of a variety of errors.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: if the error condition is not merely an echo of another PL/I condition, then restarting is often undefined. Restarting from other PL/I conditions is discussed under the individual conditions.

Data structure: the standard PL/I data structure (when not an echo).

fault_tag_1, fault_tag_3 (hardware)

Cause: the user attempted an indirect reference through a word pair containing either a fault tag 1 or a fault tag 3 modifier.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable by modifying machine conditions.

Data structure: none.

finish (PL/I)

Cause: either a run unit is being terminated because of a stop statement or the stop_run command, or the user's process is being terminated by a logout (either voluntary or involuntary) or by a new_proc command (described in the Commands manual).

Default action: returns to the point where the condition was signalled.

Restrictions: if the process is terminating because of a bump or resource limit stop, there is only a small grace period before the process is actually killed. If a user-supplied handler does not return, the process continues to run, but in some cases a subsequent process termination is fatal.

Restartability: immediately restartable.

Data structure:

```
dcl 1 finish_info          aligned based,
      2 header             aligned like condition_info_header
      2 type               char (8) ;
```

Structure elements:

header
is the same as in the information header format above.

type
indicates which situation is in effect. The values may be: run, logout, new_proc, or termsgnl.

Note: all condition handlers, whether they handle finish or not, should pass this condition on (by calling continue_to_signal_) so that all programs will be notified of the impending process, or run unit, destruction.

fixedoverflow (hardware or PL/I)

Cause: the result of a binary fixed point operation exceeded the range of the precision.

Default action: prints a message on the error_output switch and signals the error condition.

Restrictions: none.

Restartability: not restartable.

Data structure: none.

fortran_pause

Cause: the user's FORTRAN program executed a PAUSE statement.

Default action: prints a message on the error_output switch. If the process is interactive, it returns to command level. If the process is absentee, the program is automatically restarted.

Restrictions: none.

Restartability: immediately restartable.

Data structure:

```
dcl 1 pause_info        aligned based,
      2 header          aligned like condition_info_header;
```

Structure elements:

header
is the same as in the information header format above.

fortran_storage_error

Cause: an error occurred during the initialization of FORTRAN extended storage or during the creation of a COMMON block.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure:

```
dcl 1 fse_info          aligned based,
      2 header          aligned like condition_info_header;
```

Structure elements:

header
is the same as in the information header format above.

fortran_storage_manager_error

Cause: an error occurred during the assignment or creation of segments for FORTRAN extended storage (e.g. large arrays).

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure:

```
dcl 1 fsme_info        aligned based,
      2 header         aligned like condition_info_header;
```

Structure elements:

header
is the same as in the information header format above.

gate_err

Cause: the user attempted an inward wall crossing through a gate segment with the wrong number of arguments.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

Data structure: none.

illegal_modifier (hardware)

Cause: an invalid modifier appeared in an indirect word.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable by modifying machine conditions.

Data structure: none.

illegal_opcode (hardware)

Cause:	the user attempted to execute an illegal operation code.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	conditionally restartable by modifying machine conditions.
Data structure:	none.

illegal_procedure (hardware)

Cause:	the user attempted to execute a privileged instruction, or tried to execute an instruction in an invalid way.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	conditionally restartable by modifying machine conditions.
Data structure:	none.

illegal_return

Cause:	an attempt was made to restart machine conditions with invalid information.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	conditionally restartable by modifying machine conditions.
Data structure:	none.

io_error

Cause: an I/O procedure that does not return an I/O system status code received such a code from an inferior I/O procedure. The first procedure (e.g., the ioa_ subroutine) reflects the error by signalling this condition. (The ioa_ subroutine is described in the Subroutines manual.)

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure:

```
dcl 1 io_error_info      aligned based,
      2 header           aligned like condition_info_header
      2 switch           char(32),
      2 status           fixed bin(35);
```

Structure elements:

header
is the same as in the information header format above.

switch
is the name of the switch on which the I/O operation was performed.

status
is the unexpected status code received by an I/O procedure.

ioa_error

Cause: the user called an ioa_ subroutine entry point with invalid arguments. The possible incorrect calls are:

1. failed to provide a switch name for:
   ioa_$ioa_stream
   ioa_$ioa_stream_nnl

2. failed to provide a correct character string descriptor for:
   ioa_$rs
   ioa_$rsnnl
   ioa_$rsnpnnl

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

Data structure: none.

isot_fault (hardware)

Cause: an attempt was made to use an ISOT entry that had a packed pointer isot fault set. An ISOT entry contains a fault if the corresponding segment was prelinked by the prelink command before the process began, and if the segment's separate static section has not yet been referenced.

Default action: combines the static section and restarts. Prints a message and returns to command level if the static section cannot be copied.

Restrictions: In prelinked processes, a static handler is signalled for segments that do not have the copy_on_write bit set.

Restartability: conditionally restartable.

Data structure: none.

key (f) (PL/I)

Cause: the user attempted to specify an invalid key in a PL/I record I/O statement on the file f. Two examples of invalid key specifications are:

1. a keyed search failed to find the designated key
2. on output, the designated key duplicates a pre-existing key

Default action: prints a message on the error_output switch and signals the error condition. Upon return from any handler, control passes to the PL/I statement following the statement in which the condition was raised.

Restrictions: none.

Restartability: immediately restartable.

Data structure: the standard PL/I data structure.

Note: the handler can obtain the value of the invalid key by use of the onkey builtin function. The invalid key cannot, however, be corrected in the handler.

linkage_error (hardware)

Cause: the user's process encountered a fault tag 2 in a word pair. It then attempted to reference the external entry specified by the word pair and failed because either the segment was not found, the entry point did not exist in that segment, or the fault tag 2 was caused by an invalid link pair.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

lockup (hardware)

Cause: a pending interrupt has not been allowed within a set interval. This can be caused by a looping instruction pair, an infinite indirection chain, or an interrupt inhibit bit that is on for too long.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable by modifying machine conditions.

Data structure: none.

lot_fault (hardware)

Cause: an attempt was made to use a LOT entry that had a packed pointer lot fault set. A LOT entry contains a fault if the corresponding segment has been made known but has not had its linkage, if any, combined.

Default action: if the segment is an object segment, combines the linkage section and restarts; otherwise, prints a message and returns to command level.

Restrictions: In prelinked processes, a static handler is signalled for segments that do not have the copy_on_write bit set.

Restartability: conditionally restartable.

Data structure: none.

*malformed_list_template_entry_*

Cause: a compiler has generated an incorrect list initialization template for an array or an external variable.

Default action: prints a message and return to command level.

Restrictions: none.

Restartability: if restarted, the initialization that caused the fault is skipped from the point of error.

Data Structure:

```
dcl 01 condition_info        aligned,
       02 header             like condition_info_header
       02 version            fixed bin (35),
       02 variable_p         ptr,
       02 variable_end_p     ptr,
       02 template_p         ptr,
       02 template_error_p   ptr;
```

Structure elements:

header
ia the same information as in the information header format shown above.

version
a version number.

variable_p
a pointer to the beginning of the variable being initialized.

variable_end_p
a pointer to the last bit that was successfully initialized.

template_p
a pointer to the beginning of the list template initialization structure that contained the error.

template_error_p
a pointer to the list template entry that contained the error.

*

mme1, mme2, mme3, mme4 (hardware)

Cause:	the user attempted to execute the processor instruction mmeN, where N is 1, 2, 3, or 4.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	conditionally restartable by modifying machine conditions.
Data structure:	none.
Note:	the mme2 condition is handled by a static handler that calls the debug command.

name (f) (PL/I)

Cause:	an invalid identifier occurred in a PL/I get data statement on the file f.
Default action:	prints a message on the error_output switch and signals the error condition. Upon return from any handler, the invalid identifier and its associated value field are skipped.
Restrictions:	none.
Restartability:	immediately restartable.
Data structure:	the standard PL/I data structure.

no_execute_permission (hardware)

Cause:	the user attempted to execute a segment for which he did not have execute permission.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	conditionally restartable.
Data structure:	none.

no_read_permission (hardware)

Cause:	the user attempted to read from a segment for which he did not have read permission.
Default action:	prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

no_write_permission (hardware)

Cause: the user attempted to write into a segment for which he did not have write permission.

Default action: prints a message and returns to command level except when the condition is handled by a static handler or the segment has the copy_on_write bit ON, in which cases a copy is made, given write access, and given the segment number of the original segment. In the latter case, the program is restarted automatically.

Restrictions: In prelinked processes, a static handler is signalled for segments that do not have the copy_on_write bit set.

Restartability: conditionally restartable.

Data structure: none.

not_a_gate (hardware)

Cause: the user attempted to call into a gate segment beyond its call limiter; i.e., beyond the upper bound of the transfer vector in a gate.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

not_in_call_bracket (hardware)

Cause: the user attempted to call a segment from a ring not within the segment's call bracket.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

not_in_execute_bracket (hardware)

Cause: the user attempted to execute a segment from a ring not within the segment's execute bracket.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

not_in_read_bracket (hardware)

Cause: the user attempted to read a segment from a ring not within the segment's read bracket.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

not_in_write_bracket (hardware)

Cause: the user attempted to write into a segment from a ring not within the segment's write bracket.

Default action: prints a message and returns to command level except when the segment has the copy_on_write bit ON, in which case a copy is made, given write access, and given the segment number of the original segment. In the latter case, the program is restarted automatically.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

null_pointer

Cause: the user attempted to use a null pointer; i.e., a pointer with a segment number of -1 (2's complement).

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable by modifying machine conditions.

Data structure: none.

op_not_complete (hardware)

Cause: the processor has detected an inconsistency in its internal state.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable by modifying machine conditions. Upon return to the signalling procedure, the processor attempts to continue execution at the point where the op_not_complete condition was detected. The processor usually continues execution correctly but the machine state might be such that continued execution is at the user's risk. This condition is often signalled as a result of attempting to restart machine conditions that are unrestartable.

Data structure: none.

out_of_bounds (hardware)

Cause: the user attempted to refer to a location beyond the end of the segment specified.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable.

Data structure: none.

overflow (hardware or PL/I)

Cause: the result of a floating-point computation had an exponent exceeding 127.

Default action: prints a message on the error_output switch and signals the error condition.

Restrictions: none.

Restartability: conditionally restartable by modifying machine conditions.

Data structure: none.

packed_pointer_fault (hardware)

Cause: an attempt was made to load a packed pointer value containing an invalid bit offset (that is, a bit offset in the range from 60 to 77 octal). The lot_fault and isot_fault are special cases of packed_pointer_fault; the packed_pointer_fault condition is signaled only for those packed_pointer_fault faults that are neither lot_fault nor isot_fault. The system uses particular invalid packed pointer bit offsets as a mechanism for implementing lot_fault and isot_fault.

Default action: prints a message and returns to command level.

Restrictions: none

Restartability: conditionally restartable

Data structure: none

page_fault_error (hardware)

Cause: the normal paging mechanism of the Multics supervisor could not bring a referenced page into memory because the storage system device containing the page could not be read due to a hardware error that could not be corrected by the error correction mechanism.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: immediately restartable. (Sometimes retrying succeeds. However, the system already retried several times before signalling this condition.)

Data structure: none.

parity (hardware)

Cause: the process attempted to refer to a location in memory that has incorrect parity. This condition is a hardware error.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

Data structure: none.

*pascal_error_*

Cause: pascal_error_ condition is signaled when any error is detected by pascal runtime environment.

Default action: prints a message on error_output and returns to command level.

Restrictions: none.

Restartability: never restartable.

Data structure:

```
1 pascal_error_info,
  2 length              fixed bin,
  2 version             fixed bin,
  2 action_flags,
    3 cant_restart      bit(1) unal,
    3 default_restart   bit(1) unal,
    3 reserved          bit(34) unal,
  2 string              char(256) var,
  2 status_code         fixed bin(35);
```

program_interrupt

Cause: the user issued the program_interrupt (pi) command (described in the Commands manual) for the express purpose of signalling this condition. The condition is used by several commands to return to their internal request level (waiting for the next request) after the previous request is aborted either by an unexpected fault or by the user issuing a quit signal (pressing the appropriate key on the terminal, e.g., ATTN, BRK, etc.).

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: immediately restartable.

Data structure:

```
dcl 1 program_interrupt_info   aligned based
                               (program_interrupt_info_ptr),
      2 header                 aligned like condition_info_header,
      2 default_handler_restarted_this_signal     bit(1) aligned;
```

Structure elements:

header
is the same as the information header format above.

default_handler_restarted_this signal
is set on by the system default handler, default_error_handler_, if it catches a program_interrupt signal. If a handler sets this bit and restarts the condition signal, the program_interrupt command prints the message:

```
program_interrupt: There is no suspended
invocation of a subsystem that supports this command.
```

and returns to command level. If a handler restarts the condition without setting the flag, it restarts execution by calling the start command.

Note: the any_other condition handlers should pass this on.

quit

Cause:	an interactive user has requested a quit, e.g., by issuing the quit signal.
Default action:	prints "QUIT" on the terminal, aborts any pending terminal I/O activity, reverts the standard I/O attachments to their default settings, and establishes a new command level saving the current stack history.
Restrictions:	none. But, in general, the user's programs should not handle the quit condition since this condition is normally intended to bring the process back to command level. Certain subsystems can, for various reasons, still choose to make use of the quit condition; but most programs should, instead, use the program_interrupt condition as described above.
Restartability:	immediately restartable.

Data structure:

```
dcl 1 quit_info              aligned based,
      2 header               aligned like condition_info_header,
      2 switches             aligned,
        3 reset_write        bit(1) unaligned,
        3 ips_quit           bit(1) unaligned,
        3 reconnection_quit  bit(1) unaligned,
        3 pad                bit(33) unaligned;
```

Structure elements:

header
: is the same information as in the information header format shown above. The version number is one.

reset_write
: indicates whether or not a resetwrite control order is performed. If it is "1"b, the order is performed; otherwise it is not performed.

ips_quit
: is "1"b if this condition results from a quit IPS signal (i.e., the user pressed the QUIT or BREAK key on the terminal).

reconnection_quit
: is "1"b if this quit was signaled because the user's primary login channel terminal was reconnected to the process (i.e., the user issued the "connect" login request).

This page intentionally left blank.

Note: The data structure is supported by the default handler. However, it is not generated by the issuance of a quit signal. A program that wishes to simulate the effect of the quit signal may signal quit and optionally include this data struchuhe. If the data structure is not present, the default action described above will occur. If the data structure exists, reset_write being "1"b provokes the above described action. The signalling program may choose to inhibit the abortion of pending terminal write activity by setting reset_write to "0"b.

record (f) (PL/I)

Cause: a PL/I read statement on the file f read a record of a size different from the variable provided to receive it.

Default action: prints a message on the error_output switch and signals the error condition. Upon return from any handler, data is copied from the record to the variable by a simple bit string copy as though both were the length of the shorter.

Restrictions: none.

Restartability: immediately restartable.

Data structure: the standard PL/I data structure.

record_quota_overflow

Cause: the user attempted to increase the number of records taken up by the segments inferior to a directory to a number greater than the secondary storage quota for that directory.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: conditionally restartable. (More records must be made available, either by deleting segments or moving more quota into the directory.)

Data structure: none.

return_conversion_error

Cause:	either a return statement returned a value when the procedure was invoked as a subroutine, or a return statement or end statement did not return a value when the procedure was invoked as a function, or a return statement returned a value whose data type was unsuitable for the entry by which the procedure was invoked.
Default action:	prints a message on error_output and returns to command level.
Restrictions:	none.
Restartability:	not restartable.
Data structure:	none.

seg_fault_error

Cause:	the user attempted to use a pointer with an invalid segment number. This situation arises when a segment is deleted or terminated after the pointer is initialized, the pointer is not initialized in the current process, or the user has no access to the segment.
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	conditionally restartable.
Data structure:	none.

size (PL/I)

Cause:	some value was converted to fixed-point with a loss of one or more high-order bits or digits.
Default action:	prints a message on the error_output switch and signals the error condition.
Restrictions:	none.
Restartability:	not restartable.
Data structure:	the standard PL/I data structure.

storage (hardware or PL/I)

Cause:	either a reference was made beyond the maximum length of the stack or the PL/I system storage has insufficient space for an attempted allocation.
Default action:	prints a message on the error_output switch and signals the error condition. Upon a normal return the reference or allocation is retried. In the stack case, the system automatically extends the maximum length of the stack before the condition is signalled. (If the maximum length cannot be extended, the process is terminated.)
Restrictions:	none.
Restartability:	immediately restartable in stack case; conditionally restartable in PL/I case.
Data structure:	none.

store (hardware)

Cause:	an out_of_bounds error occurred while operating in BAR mode, or the user referred to a nonexistent memory (e.g., by attempting to read a clock on the memory).
Default action:	prints a message and returns to command level.
Restrictions:	none.
Restartability:	not restartable.
Data structure:	none.

stringrange (PL/I)

Cause:	the substr pseudovariable or builtin function specified a substring that is not in fact contained in the string specified.
Default action:	prints a message on the error_output switch and signals the error condition.
Restrictions:	none.
Restartability:	not restartable.
Data structure:	the standard PL/I data structure.

stringsize (PL/I)

Cause: a string value was assigned to a string variable shorter than the value.

Default action: returns to the point where the condition was signalled, causing a truncated copy of the string value to be assigned to the string variable.

Restrictions: none.

Restartability: immediately restartable.

Data structure: the standard PL/I data structure.

*sub_error_*

Cause: a subroutine has detected an error situation for which it wants to signal a condition, often with the possibility of continuing, rather than returning immediately with a status code. The sub_err_ subroutine (described in the Subroutines manual) signals this condition.

Default action: prints a message and returns to command level; however, the condition name printed is not sub_error_ but the module name from the data structure.

Restrictions: none.

Restartability: immediately restartable, conditionally restartable, or not restartable depending on the particular situation and how the action flags in the data structure are set.

Data Structure:

```
dcl 1 sub_error_info      aligned,
      2 header            aligned like condition_info_header,
      2 retval            fixed bin(35),
      2 name              char(32),
      2 info_ptr          ptr;
```

Structure elements:

header
is the same as in the information header format above. See sub_error_info.incl.pl1.

retval
indicates what action to take upon return from the handler. This datum may be changed by the handler. The meaning of particular values depends on the module signalling the condition.

name
is the name of the module signalling the condition. Within a subsystem, all calls to sub_err_ should use the generic name of the subsystem.

info_ptr
points to more information about the condition. The content and format of this information depend on which module signalled the condition.

subscriptrange (PL/I)

Cause:	the value of a subscript lies outside the range of values declared for the bounds of the dimension to which it applies.
Default action:	prints a message on the error_output switch and signals the error condition.
Restrictions:	none.
Restartability:	not restartable.
Data structure:	the standard PL/I data structure.

*sus_*

Cause:	the process is being suspended by the answering service, as a result of either an operator command, an FNP crash, or a phone line hangup.
Default action:	the handler goes blocked (i.e. causes the process to stop executing) and waits for a signal from the answering service indicating that the process has been released and can resume execution. If the process is interactive, quit is signaled in the process when the default is released.
Restrictions:	there is a site-settable cpu time limit imposed on a suspended process. If the process does anything other than going blocked, it will probably use up this time, and will then be destroyed.
Restartability:	immediately restartable.
Data structure:	none.

Note: this condition is normally handled by a static handler and, therefore, the stack is never searched.

*system_shutdown_scheduled_*

Cause: the process is being notified that the scheduled system shutdown time has been set or changed.

Default action: the handler accepts the message without taking the action and returns to the point at which the process was interrupted.

Restrictions: none.

Restartability: immediately restartable.

Data structure: none.

timer_manager_err

Cause: the timer_manager_ subroutine is malfunctioning due to damaged static storage or incorrect operation of interprocess communication in the user's process; or the user is attempting to call entries other than timer_manager_$sleep and timer_manager_$sleep_lss while in a ring other than the initial ring.

Default action: prints a message and returns to command level.

Restrictions: the user should only attempt to handle this in a handler for the any_other condition.

Restartability: not restartable.

Data structure: none.

*transaction_bj_full_*

Cause: the user's transaction attempted to write into a before journal but there is no room in the journal. The transaction must be aborted.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

Data structure:

```
dcl 1 transaction_bj_full_info aligned based
                                (transaction_bj_full_info_ptr),
      2 header                 aligned like condition_info_header;
```

Structure elements:

header
: is the same as in the information header format above; header.version must be equal to TXN_DEADLOCK_INFO_VERSION_1, declared in dm_txn_deadlock_info.incl.pl1.

*transaction_deadlock_*

Cause:
: the user's transaction is involved in a deadlock situation, meaning that two or more transactions are waiting on each other to release a lock in a fashion that will have them all waiting forever. This transaction has been selected to resolve the deadlock by aborting or rolling back itself.

Default action:
: prints a message and returns to command level.

Restrictions:
: none.

Restartability:
: not restartable.

Data structure:

```
dcl 1 transaction_deadlock_info  aligned based
                                  (transaction_deadlock_info_ptr),
      2 header                   aligned like condition_info_header,
      2 transaction_id           bit (36) aligned,
      2 file                     bit (36) aligned,
      2 control_interval         fixed bin (27) aligned;
```

Structure elements:

header
: is the same as in the information header format above; header.version must be equal to TXN_DEADLOCK_INFO_VERSION_1, declared in dm_txn_deadlock_info.incl.pl1.

transaction_id
: is the identifier of the transaction involved in a deadlock situation.

file
: is the unique identifier of the file for which the transaction was attempting to acquire a lock.

control_interval
: is the number of the control interval for which the transaction was attempting to acquire a lock.

*transaction_lock_timeout_*

Cause: the user's transaction timed out waiting for a lock.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

Data structure:

```
dcl 1 txn_timeout_info      aligned based (txn_timeout_info_ptr),
      2 header              aligned like condition_info_header,
      2 transaction_id      bit (36) aligned,
      2 file_uid            bit (36) aligned,
      2 control_interval    fixed bin (27) aligned;
      2 give_up_time        fixed bin (71);
```

Structure elements:

header
: is the same as in the information header format above; header.version must be equal to TXN_TIMEOUT_INFO_VERSION_1, declared in dm_txn_timeout_info.incl.pl1.

transaction_id
: is the identifier of the transaction involved in a timeout situation.

file_uid
: is the unique identifier of the file for which the transaction was attempting to acquire a lock.

control_interval
: is the number of the control interval for which the transaction was attempting to acquire a lock.

give_up_time
: is the clock reading when the lock timeout occurred.

transmit (f) (PL/I)

Cause: a value was incorrectly transmitted between storage and the data set corresponding to the file f. In the case of list-directed input, the condition is signalled after each assignment by the get statement of a value that might have been in error due to the bad input line.

Default action: prints a message on the error_output switch and signals the error condition. Upon return from any handler, the program continues from the point of detection as though the transmission had been correct.

Restrictions: none.

Restartability: immediately restartable.

Data structure: the standard PL/I data structure.

*trm_*

Cause: the process is being destroyed by the answering service, as a result of an operator command or a bump caused by load control or accounting limits.

Default action: the handler executes epilogue handlers and signals the finish condition, just as the logout and new_proc commands do.

Restrictions: there are site-settable cpu and real time limits imposed on a process that has been sent the trm_ signal. If an epilogue handler or finish condition handler attempts to perform any large computation, these limits will probably be exceeded, and the process will be destroyed.

Restartability: immediately restartable.

Data structure: none.

Note: this condition is normally handled by a static handler and, therefore, the stack is never searched.

truncation (hardware)

Cause: the user executed an extended instruction set (EIS) instruction to move string data and the target string was not large enough to contain the source string.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: immediately restartable.

Data structure: none.

undefinedfile (f) (PL/I)

Cause: an attempt to open the PL/I file f failed.

Default action: prints a message on the error_output switch and signals the error condition.

Restrictions: none.

Restartability: not restartable.

Data structure: the standard PL/I data structure.

undefined_pointer

Cause: The user attempted to use a pointer with a segment number of -3 (2's complement). This may represent a value that is not defined by the program.

Default action: prints a message and returns to command level.

Restrictions: none

Restartability: conditionally restartable by modifying machine conditions.

Data structure: none.

underflow (hardware or PL/I)

Cause: the result of a floating-point computation had an exponent less than -128.

Default action: prints a message on the error_output switch and returns.

Restrictions: none.

Restartability: immediately restartable.

Data structure: none.

Note: before the underflow condition is signalled the hardware register containing the floating-point value in question is set to zero.

unwinder_error

Cause: the user attempted to perform a nonlocal transfer to an invalid location.

Default action: prints a message and returns to command level.

Restrictions: none.

Restartability: not restartable.

Data structure:

```
dcl 1 unwinder_error_info aligned,
      2 header            aligned like condition_info_header
      2 invalid_label     label;
```

Structure elements:

header
is the same as in the information header format above.

invalid_label
is the invalid label to which the transfer was attempted.

zerodivide (hardware or PL/I)

Cause:	the user attempted to divide by zero.
Default action:	prints a message on the error_output switch and signals the error condition.
Restrictions:	none.
Restartability:	not restartable.
Data structure:	the standard PL/I data structure, if not detected by hardware the.

NONLOCAL TRANSFERS AND CLEANUP PROCEDURES

Many languages provide the ability to perform nonlocal transfers. In Multics, this is a facility by which the currently executing procedure activation can transfer to a location in an earlier existing procedure activation and, as a consequence, abort all activations descendant from the earlier activation. Programmers of certain types of procedures might wish to have these procedures establish a set of code to be executed if an activation of one or more of these procedures is aborted in this manner. An example of such a procedure is a program that references static data that must be reset so that the procedure can be reentered. This function of executing predefined code when an activation is aborted by a nonlocal transfer is termed cleaning up. The code for cleaning up is contained in an on unit for the cleanup condition.

Some other commonly executed actions of this type are freeing storage allocated within the program and not needed afterwards, terminating segments initiated while running the program, and releasing temporary segments.

An on unit for cleanup is established and reverted in the same way as any other condition. Unlike other conditions, however, there is no condition information associated with the cleanup condition. The cleanup condition is signalled if the establishing block activation is aborted by a nonlocal transfer. In this case, the cleanup on unit is automatically reverted when it returns to its caller; any_other handlers are not invoked for the cleanup condition.

EPILOGUE HANDLING

At the end of a run unit or process, after the finish condition is signalled, a system program is called to close files. This period in a process is called the epilogue and the program an epilogue handler. At the end of a run unit, only files opened by language I/O statements are closed by the system. At the end of a process, all files and I/O switches are closed, including those for terminal I/O.

If a user subsystem, language runtime, etc. would like to gain control of the process during the epilogue, the subroutine add_epilogue_handler_ may be called at any time during the process with the entry point of a procedure to be invoked (see the writeup in the Subroutines manual). The epilogue handlers are called before any remaining I/O switches are closed.

FAULTS

There is a class of unusual occurrences that are detected by the Multics hardware processor. These occurrences are called faults and are a subset of the set of occurrences that cause the system to invoke the condition mechanism. They are, therefore, also included under "List of System Conditions and Default Handler" above. (See also "Simulated Faults" below.)

Simulated Faults

By convention, three segment numbers are reserved for software-simulated faults. The segment numbers are dummies; in other words, no Multics segment ever has them. Any attempt to reference these segment numbers results in the out-of-bounds subcondition of the access violation fault. When this fault occurs, the supervisor fault interceptor signals (in the ring where the fault occurred) the appropriate fault. To compare a pointer to one of these special pointers, it is inadvisable to use a PL/I language test because that compares the offset as well as the segment number. One way to compare the segment number is to use the nonstandard Multics pointer built-in function to set the offset to 0. For example:

```
if pointer (ptr_to_test, 0) = baseptr ( -2) then...,
```

NULL POINTER

The segment number -1 (in 2's complement form, with the octal equivalent "77777"b3) is reserved for the null pointer. This is accessible as the null built-in in PL/I. Any reference to this segment number signals the null_pointer condition.

PROCESS TERMINATION FAULT

The segment number -2 (in 2's complement form, with the octal equivalent "77776"b3) is reserved for the process termination fault. Any reference to that segment number causes the referencing process to be terminated. This is accessible in PL/I by using baseptr (-2).

UNDEFINED POINTER FAULT

The segment number -3 (in 2's complement form, with the octal equivalent "77775"b3) is reserved for the undefined pointer fault. Any reference to that segment number causes the undefined_pointer condition to be signalled.

SECTION 8

BACKUP

The Multics backup systems augment the reliability of the online storage system. They ensure that user segments and directories can be recovered if they are destroyed due to system failure or user error.

The backup system performs the following two functions:

1. dumping

   The backup mechanism searches out, selects, and copies (dumps) onto tape segments and directories from the Multics storage hierarchy. The frequency of dumping and the length of time for which tapes are kept are determined at individual locations.

2. recovery

   Reloading is a global recovery of segments and directories that have been dumped. Retrieving is the recovery of individual segments and directories that can occur during normal Multics operation. The entire contents of the online storage system can be reloaded after a system crash so that operation of the system can resume.

There are two major Multics backup systems, hierarchy and volume. The hierarchy system tree-walks the hierarchy to locate the data it must dump, while the volume system scans the physical volumes used by the storage system. The goals of both systems are the same, but the mechanism, cost, and benefits differ. For more detailed information see the *Multics System Maintenance Procedures* manual, Order No. AM81.

DUMPING

The dumping mechanism operates in three different modes: incremental, consolidated, and complete. These modes are distinguished by three different criteria used to select segments and directories for dumping. What is dumped is site-controllable. Usually, only two subdirectories of the root directory are not searched. One of these, `>system_library_1`, is always recreated by a Multics bootload and therefore does not require the services of backup. Parts of the hardcore system, plus that part of the command system needed during reloading, are contained in `>system_library_1`. The other subdirectory, `>process_dir_dir`, contains only per-process information that is temporary in nature and hence also does not require the services of backup. Libraries that never change need not be included in the search route for incremental dumps (defined below). All other sections of the hierarchy should be included in the search route of the backup system.

Multiple dumper processes registered as SysDaemon, Daemon, or both, are allowed.

Incremental Dumps

Incremental dumping is the principal technique used to keep the backup systems abreast of changes to online storage. It is the purpose of an incremental dump to discover modifications to online information not reflected in backup tape storage. The incremental dump, starting from a specified search node, locates and dumps all segments and directories modified more recently than they have been dumped. The net effect of the incremental dumping scheme is to limit the amount of information that can be lost to those modifications that have occurred since the last incremental dump.

Incremental dumping is triggered periodically by the alarm clock timing mechanism. In order to minimize the time span during which modifications to online storage can go unnoticed by the backup system, incremental dumps should be produced frequently. On the other hand, because the backup daemon competes with ordinary users and exerts a considerable drain on system resources, it becomes economically desirable to lower the frequency of incremental dumps. Therefore, the interval between the incremental dumps at an installation is chosen as a compromise between these two considerations. This does not imply that an incremental dump will necessarily finish its search within a single time interval. In fact, if the incremental dumper is given no scheduling advantage, several intervals might be required to complete an incremental dump during hours of heavy system load. If an incremental dump is not completed before the next incremental dump is scheduled to begin, the "next" dump is deferred until the prior incremental dump is completed.

The backup system does not guarantee that segments are dumped in a consistent state. For example, it is possible that while the incremental dumper is dumping a segment, another process might be writing into that same segment. Thus, an inconsistent copy of a segment might be produced. However, the modifications that cause a segment to be inconsistent also cause another dump of the segment to be produced on the next pass of the incremental dumper. Therefore, unless the system crashes before the next incremental dump, a consistent copy is eventually produced.

Consolidated Dumps

A consolidated dump, starting from a specified search node, locates and dumps segments and directories that have been modified after some specified time in the past. For example, an installation might choose to run a consolidated dump every midnight to dump all segments and directories modified since the preceding midnight; i.e., since the preceding consolidated dump. Since a consolidated dump catches modifications accrued over a period of time encompassing many incremental dumps, it effectively consolidates the most recent information from a group of incremental tapes and thereby facilitates the reloading of this information by decreasing the number of tapes that must be processed. Also, since tape is susceptible to operational, hardware, and software errors, a consolidated dump provides the installation with a second tape copy of the segments and directories dumped during an incremental dump.

Complete Dumps

A complete dump, starting from a specified search node, dumps every segment and directory in the storage system without regard for modification time. Unlike incremental and consolidated dumps that attempt to keep the backup tapes up-to-date with the contents of the storage system, complete dumps are somewhat different in purpose and follow a more leisurely schedule.

A complete dump establishes a checkpoint in time, essentially a snapshot of the entire Multics storage hierarchy. If it should ever become necessary to recover the entire contents of online storage, then the tape with the most recent complete dump marks a cutoff point beyond which no older backup tapes need be inspected.

The high production rate of incremental and consolidated tapes makes the retention of these tapes for long periods of time impractical. Therefore, incremental and consolidated tapes are kept for some short time, perhaps 3 weeks. Complete backup copy tapes are retained for a longer time, perhaps 6 months, with the exception of one complete dump tape per month that might be held for a period of 1 year.

RECOVERY

When a user notices that a segment or directory has been lost or damaged, he can submit a request to the Multics operations staff for that segment or directory to be retrieved from a backup tape. The problem he faces is determining which backup dump operation produced the tape copy of the segment or directory he wishes to retrieve. Usually the most recently produced copy is wanted. In the case of a damaged segment, however, the damaged version is likely to have been dumped as well, and hence the most recent tape copy may not be wanted. Hopefully, a user knows approximately when his segment was lost or damaged. Also, he should remember if the segment has been recently modified. Using these two pieces of information, he can make a reasonable guess as to when the last usable copy of the segment was online.

Once an estimate has been made as to the time frame, this estimate can be verified by examining the corresponding hierarchy dump map. This operation is automatic for volume recovery, although the user can still specify the time frame if desired. The map indicates the tape reel on which the dump was written. A feature of the dump map that is sometimes helpful is the printing of the date-time-dumped attribute for the segment, which effectively points to the next most recent tape copy of the segment.

The user can specify that a single segment, a directory without its subtree, or a directory including its subtree be recovered.

In special cases, a user can specify that a segment or, for hierarchy recovery only, a directory be reloaded with a different pathname. A single segment or a directory without a subtree can be relocated at any point in the storage system hierarchy. For hierarchy recovery only, a directory subtree can be relocated at any point in the hierarchy.

For more information on volume retrieval, the reader is referred to the description of enter_retrieval_request in the Commands manual.

SECTION 9

ADMINISTRATIVE CONTROLS

Multics administrative control facilities decentralize the tasks involved in managing system resources. These facilities are available only to a group of privileged users who have been given system or project administration responsibility. They provide a resource management interface to the user that matches the policy of the installation.

ADMINISTRATIVE HIERARCHY

Administration of Multics resources is organized into a hierarchy of responsibility; this hierarchy maps onto the storage system hierarchy. Figure 9-1 shows a typical administrative hierarchy. At the top are the system administrators, who allocate resources among the projects within the system. Project administrators allocate these resources among users assigned to the project. Each user, in turn, can control the use of his resources through storage management and access control facilities.

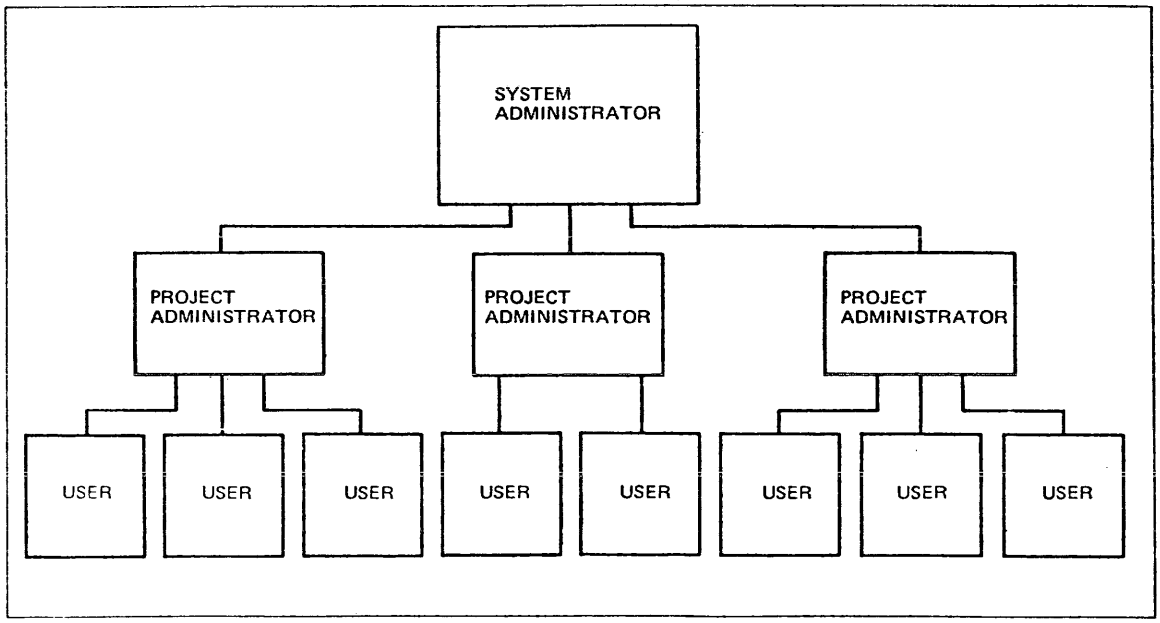


Figure 9-1. Multics Administrative Hierarchy

System Administrators

System administrators are responsible for establishing the system configuration, for allocating resources among projects, for maintaining system security, and for keeping up-to-date accounting records. System administrators can delegate some of these functions to project administrators and can also assume project administration responsibility. The functions generally performed by system administrators are summarized below.

1. Resource Management

   a. establish resource-allocation groups among users (load control groups and work classes)

   b. establish priorities for allocation of peripheral devices

   c. establish the quantity of resources available to users at a particular time (system configuration scheduling)

2. Accounting Procedures

   a. up-to-date records of system usage so that users are billed for resources used

   b. set prices for system resource usage

   c. generate usage reports and bills (these can be issued at any time without interrupting Multics service)

3. Security Control

   a. register projects, users, and initial user passwords

   b. define rings of access in which a project can operate

In addition, system administrators select the operating schedule and system tuning parameters that best serve the needs of the service.

Project Administrators

Project administrators control the allocation of resources among users assigned to particular projects. A project can have from one to four administrators, or the system administrator can serve as project administrator.

Project administrators establish the operating environment in which the user's process can run. They can choose the initial procedure, the user's home directory, the terminal outer module, and several internal table sizes. They can also specify that the user will be logged in as part of a special subsystem with limited or no access to the standard Multics environment. Project administrators can restrict the user's ability to modify per-process parameters at login time. Furthermore, project administrators can, within limits set by the system administrator, specify the user process's access authorization and ring of execution.

Project administrators can limit user expenditures by establishing a total dollar limit, a monthly limit, a daily limit, a per-shift limit, or any combination of these. Resource management programs are provided by the system, but project administrators can also write their own programs to manage these tasks.

Users

On Multics, the term user applies both to people and logical entities, such as daemons, who have the ability to log in. This ability requires that the user be registered by a system administrator. Each registered user is identified by two items of information: a Person_id, which uniquely identifies the person or entity; and a Project_id, which identifies the user as part of a particular resource-control group. A person or entity can be associated with more than one project. Each combination of Person_id and Project_id identifies a separate user, who is allocated resources according to the identification used at a particular login.

Several attributes are recorded for each registered Person_id in system tables. These include a personal password, default initial ring, default authorization, and default Project_id. Other attributes are maintained on a per-user basis. These include home directory pathname, initial procedure, preemption grace time, initial and maximum ring, process table sizes, terminal outer module pathname, resource consumption limits, and attribute flags that indicate special properties of the user such as his load control group status and permission to change his initial procedure.

A user can perform a variety of administrative functions that help him to manage his own resources. Some of these are listed below.

- check his own use of resources such as total dollar usage, processor time used, I/O and absentee usage
- specify who can have access to his segments and directories and the type of access for each; access is granted or revoked by command and the user's specification takes effect immediately
- create directories under his home directory at his own convenience
- change his password

- change his default project
- specify how control will be transferred to his process

ADMINISTRATIVE CAPABILITIES

The following paragraphs describe some of the major administrative capabilities provided on Multics; these comprise a sizeable subsystem. For a complete description of Multics administrative capabilities, see the *Multics System Administration Procedures* manual, Order No. AK50, and the *Multics Administrators' Manual -- Project Administrator,* Order No. AK51.

Pricing

System administrators set prices for each type of system resource that a user may consume. The user can check on his current consumption of resources by issuing the resource_usage command.

INTERACTIVE AND FOREGROUND ABSENTEE USAGE

Three kinds of usage are measured for user processes logged in from a terminal or logged in as foreground absentee. These are:

- CPU time
- memory units
- connect time

Usage of these resources is recorded and charged for separately on each shift. The shift schedule is established by the system administrator. Up to eight shifts can be defined; the administrator specifies the shift for each half-hour period of the week.

BACKGROUND ABSENTEE USAGE

User processes created as a result of a user's enter_abs_request command can perform background, batch, deferred, or periodic processing. Two kinds of usage are measured:

- CPU time
- memory units

Usage of these resources is recorded and charged for separately in each queue. Usually, four queues are provided. System administrators can also specify which queues run on which shift.

I/O DAEMON USAGE

The Multics I/O daemon manages the use of remote I/O devices and the local line printer and card punch. User requests for I/O operations are placed in queues by the enter_output_request, dprint, or dpunch commands. Each request has a priority number and request type that are used to determine the queue in which it will be placed. Most installations provide four priority queues for I/O daemon processing for each request type, and set different prices for each. Usage of the printer is measured in lines and pages; usage of the punch is measured in cards.

OTHER CHARGES

System administrators also set prices for several other resources. Some examples are:

- storage (in page-seconds)
- special channels (in microseconds)
- registration fee (by month)

Apportioning System Capacity

System administrators define two major groupings of users for purposes of controlling the sharing of system resources. These are load control groups and work classes. A load control group is a set of projects that share a guaranteed access to the system. A work class is a set of users to whom a certain percentage of CPU time has been allocated.

LOAD-CONTROL GROUPS

Each load control group has a quota of primary load units representing a number of users within the group who will always be able to log in. Users in excess of the primary quota are assigned secondary status and will be allowed to log in if the system is not full; these users can be preempted if a primary user wants to log in when the system is full. If the group's primary quota is full and the system is full, a user from the group cannot log in unless he has been given the right, by the project administrator, to preempt another user. Every user is assigned a minimum time, at login, during which he is protected from preemption. The status of each user within the load control group and the minimum time that each user will be protected from preemption are established by the project administrators responsible for projects in a particular load control group.

WORK CLASSES

System administrators can divide the system's CPU resources into pools called work classes. A work class can be an entire load control group, a single user, or a class of users such as all I/O daemons or all absentee jobs in a given queue. The membership in a particular class and the percentage of CPU time assigned to a particular class can be changed at any time by a system administrator.

Access Control

System administrators are responsible for system security and maintain it by controlling user access to a variety of system resources. The same kind of access control lists are used for system access control as are used for controlling access to segments and directories in the storage system.

GATE ACCESS

System administrators control access for many privileged or semiprivileged gate segments that allow the user to request some specific supervisor action. Each of these gates has an access control list that defines the set of users who can call it. For example, one gate allows users to read and write tapes using the I/O system. If the system administrator denies a particular project access to this gate, no user on the project can read or write a tape.

Project administrators can arrange for all users on their project to log in into a higher ring than usual, and then provide gates from this higher ring to selected services in the standard user ring. Thus, project administrators can exercise complete control over which parts of the Multics environment a user process can access.

DEVICE ACCESS

Each external I/O device attached to the system through the I/O system has an access control segment (ACS) associated with it. Thus, system administrators can specify which devices are available to all users (e.g., tape drives) and which devices are reserved for system processes only (e.g., printers).

VOLUME ACCESS

Each private logical volume registered by the storage system has an access control segment (ACS) associated with it. Unless a user has rw access to the ACS associated with a logical volume, he will be unable to attach the volume and thus unable to use any segments on the volume. If the user has e access to the logical volume, he is permitted to manipulate the quota specifications for subtrees on the logical volume.

ABSENTEE AND DAEMON QUEUES

System administrators can modify the access control lists that permit users to place requests in the absentee or I/O daemon queues in order to deny certain classes of users access to these facilities entirely.

Storage Quota

The Multics quota facility allows the creator of a directory to specify the maximum number of records that can be occupied by segments and directories inferior to the quota directory. Quotas can be moved (delegated) from a superior directory to an inferior directory.

System administrators are responsible for the distribution of quota on the public logical volumes. They delegate the responsibility for quota management to the project administrators and other managers by moving quota down to directories (such as project directories) to which the quota managers have modify access.

Other logical volumes are managed in a similar fashion, but instead of the system administrators controlling the total amount of quota and its initial distribution, these privileges are given to any user who has e access to the logical volume.

SECTION 10

MULTICS DATA MANAGEMENT

This section presents an overview of Multics Data Management comprising seven major subsections, as follows:

- Introduction
- Data Management Files
- Using MRDS with Data Management
- Data Storage and Retrieval Services
- Integrity Services
- Administering Data Management
- Command Level Interface to Data Management

INTRODUCTION

Multics Data Management provides services for storing, accessing, and protecting information in data management files. Data storage and retrieval services provide for space allocation, data organization, and file access. Integrity services offer protection in a transaction-based environment by maintaining database consistency, controlling concurrent access to the database, and providing the means to recover from various failures.

The Data Management System (DMS) is based on a layered architecture, in which requests are made through the top-most layer and passed down from level to level to the Multics supervisor. Software modules have specific responsibilities on a given level, with no knowledge of, or responsibility for, what takes place above or below them. The layers of the architecture are as follows:

- The external access layer provides logical access to data. The Multics Relational Data Store (MRDS) can be viewed as an interface to this layer. The relation, record, and index managers are the software modules that support this layer.

- The collection access layer provides logical storage management. Its services include allocating and freeing the individual logical units of data that make up a file. The collection manager supports this layer.

- The file access layer is the interface between the higher layers in the architecture and the low-level Multics functions that manage the physical representation of data. This layer handles all requests for storing and retrieving data and is responsible for converting logical addresses to physical addresses. The file manager supports this layer.

The architectural framework of DMS provides several advantages over other storage methods. The layered structure ensures compatibility with future software and product enhancements because users (programmers and other system modules) always "see" a consistent interface. The origin of database requests is masked so that whether from MRDS, another access method, or another machine (given the networking capability to do so), each request "looks" the same to the layered software. Expandability has been built in to the system so that when a facility such as distributed systems architecture (DSA) becomes available, it can be used by existing database applications with virtually no redesign or reimplementation effort.

The modular design enables the system to make use of the same software in all situations, so that a service is always performed in the same manner by the same module no matter who the caller is. Recovery, for example, can be instigated by different modules at different points of processing, but the rollback procedure invoked is always the same. Calls to begin and end transactions can be specified by user programs, by users interactively, or by the system itself, but the calls are always to the same entry points. This specialization reduces the complexity of the software, making it much easier to maintain and upgrade.

Currently, Multics Data Management is intended for use with MRDS applications to allow them to benefit from transaction processing and the attendant protection features, including a guaranteed consistent database, concurrency control, and the ability to roll back database modifications.

Features and Benefits of Multics Data Management

DMS offers the following advantages:

- The contents of data management databases are protected against loss or damage due to failure, including system crash without emergency shutdown (ESD) procedures.
- Protection can be turned off at file creation to save on overhead, when data integrity is not a concern.
- The working unit of DMS is a transaction, a user-defined entity; thus, you tailor the system to your applications, not vice versa.
- Deadlocks are detected and resolved automatically so that waiting transactions can continue to run.
- Strict protocols are imposed to support concurrent access to the database without threatening data integrity.

- The same software modules and the same interfaces are always called to perform the same functions, to ensure a consistent processing environment.
- You can adjust the level of locking granularity for the same data to suit different processing conditions.
- Transactions can be started by applications programs, interactively from command level, or automatically by MRDS.
- A system default before journal can serve all DMS users, or you can create and use your own journals in your hierarchy.

In addition to the above, MRDS applications that convert from vfile_ to data management databases realize the following improvements and extensions:

- A MRDS vfile_ database converted to MRDS data management format may use significantly less disk space to store the same data.
- Tuples and their associated indices are kept consistent across failures.
- Locking occurs at a finer level of granularity to allow increased concurrency.

DATA MANAGEMENT FILES

The functional capabilities and services that Multics Data Management provides are available to applications that access specific file entities known as data management or DM files.

A DM file is an object in a Multics hierarchy, referenced by a pathname and protected by the security features inherent to the Multics file system. In the current implementation of Multics Data Management, a DM file exists as a ring 2 multisegment file, all the components of which have identical ACLs. Ring 2 is used to ensure that DM files are properly formatted and to protect against inadvertent or malicious damage.

A DM file consists of logically contiguous, fixed-size blocks called control intervals, numbered consecutively from zero. The size of a control interval, which is the unit of data transfer in Data Management, is currently fixed at 4096 bytes. The size of the file itself is limited only by the number of entries permitted in a directory.

Creating Data Management Files

There are several ways to create a DM file:

- Create a MRDS relation in DM file format (MRDS automatically calls the relation manager, as described under "Relation Manager")
- Call file_manager_$create (see "File Manager as a Direct Interface")
- Use the create_dm_file command (see "Command Level Interface to Data Management")

In creating a DM file, the caller supplies a pathname and a pointer to a file_create_info structure describing the file's attributes. Attributes include version, size, blocking factor, and whether the file is protected, i.e., whether it is entitled to integrity services. If no info structure is supplied, one is created with default values.

Data Management Files as Protected Entities

A protected DM file is entitled to integrity services, which means that its contents cannot be compromised by concurrent access and cannot be left inconsistent because of processing failures. A protected DM file can be accessed only in transaction mode, that is, a transaction must be started in the user process to access the file (this is not the case for unprotected DM files).

A DM file is protected by default at file-create time, which implies that the file protect attribute is on, and the no_concurrency and no_rollback options are off. The settings can be altered explicitly when the file is created. Turning the no_concurrency and no_rollback bits on prevents locking and recovery respectively. If the file protect bit is off, neither locking nor recovery is available, regardless of the no_concurrency and no_rollback settings.

Since file protection is optional, the overhead involved is incurred only on an as-needed basis. Those files for which protection is turned off cannot be recovered. See the description of "Integrity Services" for detailed information on file protection and the concept of transactions within Multics Data Management.

Accessing Data Management Files

Though like a segment in most respects, a DM file cannot be directly accessed by hardware address (segment number, word number) from a user program. Rather, software calls are used to open and close DM files, and to store and retrieve data. These calls are made in the form of MRDS dsl_ operations or direct calls to the file manager, depending on how the application is implemented. See the description of "Data Storage and Retrieval Services" and in particular "File Manager."

Manipulating Data Management Files

As file objects, DM files are considered to be in the same class as segments, directories, links, and multisegment files. Like these other standard entry types, DM files can be moved, copied, deleted, etc., as part of normal Multics storage system manipulation. The Multics file system utility supports use of existing file system commands to manipulate DM files by providing for uniform handling of file system entries.

Listed below are the file system commands that recognize standard entry types. Each command is described in the context of its treatment of DM files. Although all of these commands recognize standard entry types, some of them are inappropriate for use with DM files, and are so noted. For complete descriptions of these commands, see the *Multics Commands and Active Functions* manual, Order No. AG92.

delete
: results in the specified DM file being deleted, but because the delete is a protected operation, the file is not actually deleted unless the transaction commits (i.e., the effect of the delete would be rolled back if the transaction is aborted).

delete_dir
: deletes the named directory and subtree. Any DM files resident in the directory would be subject to the same conditions that apply under the delete command.

list
: lists selected information pertaining to all DM files in a single directory.

status
: prints selected detailed information on the specified DM files.

copy
: creates a new version of the specified DM file by retrieving each control interval and storing it in the new file. The two versions differ in their unique ids and in the time-last-modified stamp in each control interval.

copy_dir
: copies the specified directory and subtree to another point in the hierarchy.

move
: copies the specified DM file as with the copy command. Subsequent deletion of the original DM file is subject to the same conditions that apply under the delete command.

move_dir
: moves the specified directory and subtree to another point in the hierarchy. Subsequent deletion of DM files in the original directory is subject to the same conditions that apply under the delete_dir command.

add_name
: adds an alternate name for the specified DM file entry.

delete_name
: you cannot delete a DM file name.

copy_names
: copies all alternate names from one DM file to another, leaving all names in the original entry.

move_names
: you cannot move DM file names.

rename
: you cannot rename a DM file. Further, you are cautioned not to rename a directory containing DM files because to do so may make it impossible to recover those files in the event of failure.

list_acl
: prints the access control lists (ACLs) of specified DM files.

set_acl
: modifies the ACLs of specified DM files.

delete_acl
: removes entries from the ACLs of specified DM files.

copy_acl
: copies the access control list from one DM file to another, replacing the current ACL, if necessary.

set_ring_brackets
: modifies the first and second ring brackets for the specified DM file. This does not alter the file's implementation as a ring 2 multisegment file, which is enforced by hardware.

USING MRDS WITH DATA MANAGEMENT

The protection features of Multics Data Management integrity services are available to MRDS applications that access and modify protected DM files. This implies that the MRDS database is formatted as one or more DM files, which in turn, implies that either a new MRDS database was created as such, or an existing vfile_ database was converted to DM file format. Note that a database cannot contain both vfile_ and DM file relations.

Users of Data Management are assumed to be knowledgeable MRDS users, conversant with the *Multics Relational Data Store Reference Manual*, Order No. AW53. Also see Appendix G, "Data Management Facility Interface," in this same manual for a comparison of DM file and vfile_ usage from the perspective of a MRDS user.

Building an MRDS Data Management Database

To create an unpopulated MRDS database in DM file format, supply the appropriate data model source segment to the create_mrds_db command with the -data_management_file control argument. All specified relations will be created as protected DM files by default. You can then load the database in the usual fashion by storing tuples in the relations and creating indices as necessary. The Data Management System, specifically the relation manager, assumes responsibility for performing these functions on behalf of MRDS (see "The Relation Manager and MRDS Database Requests").

If you wish to convert an existing vfile_ database to DM file format, use the data model source for the vfile_ database (Version 4 or later) as input to the create_mrds_db command with the -data_management_file control argument. Then use the copy_mrds_data command to copy data from the vfile_ database to the DM file database. If the original data model source is not available, you can re-create it using the display_mrds_dm command with -cmdb control argument. The create_mrds_db, copy_mrds_data, and display_mrds_dm commands are described in the *Multics Relational Data Store Reference Manual*, Order No. AW53.

Using MRDS Applications with DM Files

Any retrieval or update action against a protected DM file must be performed as part of an atomic operation that has an explicitly defined beginning and end. This operation is known as a transaction and is the working unit of Multics Data Management. The concept of a transaction as perceived by Data Management is described fully under "Integrity Services" and in particular "Transactions and Database Consistency."

Unlike other systems that dictate what constitutes a transaction, Multics Data Management allows users to build their own transactions according to the requirements of the application. You decide what set of database requests comprises a working unit and "wrap" them with beginning and end points to form a transaction. The transaction manager, one of the integrity services modules, is the user interface to writing atomic operations within applications programs (see "Defining Transactions").

If you do not wish to modify your programs, a default strategy exists whereby MRDS will wrap each database operation automatically to form individual transactions and ensure integrity services for the DM files involved (see "Building Transactions in Existing MRDS Applications").

DATA STORAGE AND RETRIEVAL SERVICES

Data storage and retrieval services under Data Management include:

- Maintaining consistency between relations and their indices
- Record formatting and indexing
- Space allocation and storage management
- data management file organization and data access

The service managers that support data storage and retrieval are listed below, together with brief descriptions of the respective roles they play in administering these services. Each manager is also described in greater detail under separate heading. All managers share in translating MRDS commands and dsl_ calls into data management functional capabilities, in keeping with the design aspects of layered architecture.

- Relation manager
  serves as an interface between the database manager and other data storage and retrieval managers, in fulfilling database requests from applications programs.

- Record manager
  manages the tuples of a relation as a collection of records, chained together for efficient access.

- Index manager
  manages the indices of relations as collections of keys, maintained in sorted order.

- Collection manager
  allocates and frees control intervals within collections and manages element storage within these control intervals.

- File manager
  serves as interface between data storage and retrieval services and Multics file access and control mechanisms in handling an application's database requests.

Multics Data Management currently supports MRDS applications (including LINUS requests) that access and update MRDS databases consisting of one or more DM files. Under certain conditions, the file manager can be called directly by user applications (see "File Manager as a Direct Interface," below).

Relation Manager

In its role as an interface between the database manager (MRDS) and the data storage and retrieval services of Multics Data Management, the relation manager assumes the task of managing individual relations and their indices. MRDS is left with the responsibility of managing the global properties of all relations comprising the database and of providing the interface to user applications.

THE RELATION MANAGER AND MRDS DATABASE REQUESTS

An application program or LINUS user makes a request to MRDS to access a database. MRDS identifies the target database as a vfile_ or DM file structure and channels the request accordingly. Requests to access DM files are translated into calls to the relation manager. These calls in turn generate calls to other managers of DMS, depending on the request. In fact, a single database request usually triggers a sequence of calls between managers on each level within the DMS architecture to complete the requested activity (see the discussion of layered architecture in the Introduction). This dispersal of responsibilities simplifies the system software.

For example, a MRDS request to create a relation sets off a chain of events culminating in the creation of a relation in the form of a DM file (see below), constructed of control intervals and organized into collections into which will be stored the tuples of the relation. The relation, file, and collection managers all collaborate automatically to effect this outcome.

Similarly, a MRDS request to open a relation is translated by the relation manager into a call to the file manager to open the equivalent DM file and return a file opening id by which all subsequent calls to the file will be made until the file (relation) is closed. The relation manager's function is to interpret the MRDS request, channel it to the appropriate manager(s), and pass the result back through MRDS to the application.

Other managers of the data storage and retrieval services are described below as they interact with the relation manager and each other.

RELATIONS AND DATA MANAGEMENT FILES

A relation currently has a one-to-one correspondence with a DM file. Since there is one relation per file, the name of the relation is the same as its DM file equivalent.

All tuples of a relation are stored in control intervals and organized as a single collection of records, indexed by one or more collections of keys. These collections are organized by the collection manager and managed by the record and index managers respectively. The relation manager ensures that tuples and their indices are kept consistent across failures for protected DM files.

Record Manager

The record manager manages collections of records stored as tuples by the relation manager. These logical records may have multiple fields of varying length and mixed data types, but all records in a collection must have the same set of fields. The record manager supports operations to create and delete record collections, to create, delete, and position cursors, and to put, get, modify, and delete records.

Record search criteria, as specified through MRDS dsl_ selection expressions (-range, -select, -where), are interpreted by the relation manager and passed to the record manager to locate records by numeric positioning or a search on values. Search specifications can be absolute (from the beginning of the record collection) or relative (from a specific position within the collection maintained by a cursor).

Currently, the relation manager is the primary user of the record manager's services. The record manager itself uses the services of the collection manager to store and retrieve records in collections.

Index Manager

The index manager is used by the relation manager to store keys in an index and manage them in sorted order. It implements indices of keys as collections, which complement collections of tuples in the same relation. These keys may have multiple fields of varying length and mixed data types, but all keys in a collection must have the same set of fields. The index manager supports operations to create and destroy indices, to create, destroy, position, and copy cursors, and to put, get, and delete keys.

Key search criteria, as specified through MRDS dsl_ selection expressions (-range, -select, -where), are interpreted by the relation manager and passed to the index manager to locate keys by numeric positioning or a search on values. Search specifications can be absolute (from the beginning of the index) or relative (from a specific position within the index maintained by a cursor).

Currently, the relation manager is the primary user of the index manager's services. The index manager itself uses the services of the collection manager to store and retrieve keys in collections.

Collection Manager

The collection manager reserves control intervals for collections, allocates and frees control intervals within collections, and manages the storage of elements in these control intervals. A collection is logically a set of elements of data and physically the set of page-sized control intervals used to store the elements in a DM file. Although elements are in fact records or keys, they are viewed as simple bit strings by the collection manager.

Elements can be stored in control intervals using either of two methods: basic element storage seeks free space anywhere in the collection for the element; ordered element storage seeks free space in a specific control interval. Thus the collection manager determines physical placement of records being stored as tuples in a relation.

To perform its storage management function, the collection manager maps control interval layout within DM files. As information is manipulated in a DM file (retrieved, modified, rewritten, deleted), the collection manager adjusts element offsets, restores free space, and acquires and releases control intervals, as needed.

The record and index managers are the primary users of the collection manager's services. They know which records or keys to retrieve based on the search specifications interpreted by the relation manager. The collection manager knows where to locate the data based on the record or key identifiers supplied by the respective manager.

The collection manager itself uses the services of the file manager to do actual data storage and retrieval. The file manager is the only module that can read from or write to DM files, but only does so when the collection manager indicates which control intervals to access and where within the control intervals to get or put data.

File Manager

The file manager is the only Data Management manager that actually manipulates DM files. It is the interface between MRDS application requests (as processed by the upper layers of the DMS architecture) and the Multics file access and control mechanisms constituting virtual memory management. In this capacity, the file manager supports the following operations:

- Creates DM files
- Opens and closes DM files
- Releases logical address space
- Retrieves and stores data in DM files

Additionally, the file manager is the link to integrity services as provided by Multics Data Management.

FILE MANAGER AND DM FILE MANIPULATION

DM file creation results from a request to create a MRDS database in DM file format (see "Using MRDS with Data Management"). The request is passed to the relation manager, which generates a call to the file manager to create a DM file. Included in the call are the file's pathname and a pointer to a file_create_info structure containing the file's attributes. If the pointer is null, the file is created with default attributes (see the description of the file_manager_ subroutine). A DM file is protected by default, which means it is entitled to the protection features of integrity services. The file is created with rw access for both the caller and the data management daemon (Data_Management.Daemon).

Other operations are handled in similar fashion, with requests originating from MRDS applications being interpreted by the relation manager and then channeled through the appropriate data storage and retrieval module to the file manager. Files accessed through the file manager are subject to the same security and access control constraints applied to other file objects in the Multics storage system.

Requests to open and close relations result in calls to the file manager to open and close DM files. When it opens a DM file, the file manager returns a file-opening-id by which all future access to the file will be made, until the file is closed. Multiple openings of a file are permitted within a process, but the file must be closed the same number of times it is opened. A single close operation merely decreases the number of openings by one.

Requests from MRDS applications to read and write data in MRDS DM file databases result in calls to the file manager to get and put data in DM files. The file manager uses the file-opening-id, control interval number, byte offset, and length of the piece of data to perform these operations. This combined address is derived through the collaborative efforts of the data storage and retrieval managers, with no additional input from the MRDS application.

FILE MANAGER AND INTEGRITY SERVICES

The file manager also ensures concurrency protection and rollback capability by generating calls to the lock or before journal managers, as appropriate, when protected DM files are accessed or modified by processes executing transactions.

To protect against concurrent access, the file manager, when requested to get or put data in control intervals of protected DM files, first calls the lock manager to set the appropriate locks on the target control intervals. No data access occurs until the proper locks are set. For a complete description of locking and protection from concurrent access, see "Integrity Services."

To make locking more efficient, the file manager considers "lock advice" when opening a DM file. Lock advice is a consideration of the MRDS scope setting as declared on a relation. For a given operation, a more global lock setting might be appropriate. For example, when every control interval in the DM file is to be accessed, it would be more efficient to lock the entire file rather than each control interval individually. Lock advice serves as a guideline; the lock manager ultimately sets the requisite locks to guarantee file protection.

The file manager supports journalization to provide the rollback capability. When requested to modify control intervals of DM files (put operations), the file manager first calls the before journal manager to log before images of the affected control intervals. These images are physically written to a before journal before a DM file is modified so that the file can be rolled back in case of failure. On a rollback, the before journal manager calls the file manager to undo the modifications by writing the before images back into the DM file control intervals. For a complete description of rollback and recovery procedures, see "Integrity Services."

FILE MANAGER AS A DIRECT INTERFACE

The file manager can be used as a direct interface to make the protection and recovery capabilities of integrity services available to users who write applications using their own data storage and retrieval software.

INTEGRITY SERVICES

People and society have become almost totally dependent on the information capture and retrieval capabilities of computers. Such a dependency underscores the necessity for ensuring the reliability of information in the event of computer failures. Thus, a keystone of computer operations is to keep the information repository, the database, consistent across these failures.

Consistency is rooted in a concept of discrete processing activities known as transactions. Data that is changed as a result of the activity remains changed if the activity is successfully completed, or is restored to its original state if the activity cannot be completed. The database is never left in an inconsistent state, i.e., partially done or undone as a result of the transaction activity.

Database availability is vital in ensuring a useful database production environment, and concurrent transaction activity optimizes availability. If unchecked, however, such concurrent activity may threaten database consistency. To guard against this threat, access to data is controlled so that no transaction can interfere with another transaction's previously established claim to that data. Control over concurrent access is enforced by "locking" the data on behalf of the requesting transaction.

A failed transaction imperils database consistency and keeps locked data inaccessible to other transactions. To offset these consequences, it must be possible to recover from the failure by reversing the effects of database modifications caused by the transaction and releasing the locks held by it. Reversing the effects of modifications is known as rolling back the transaction, and is possible because of journalization, a method in which images of the data in its original state are preserved in a log called a before journal.

The concepts of database consistency, concurrent access control, and recovery constitute integrity services under Multics Data Management. The transaction manager orchestrates support of these services through the before journal, lock, and file managers. Together, these modules interact with user applications and the data storage and retrieval modules to guarantee integrity services in a transaction-driven environment that uses data management (DM) files. Support of these services is described below in greater detail.

Transactions and Database Consistency

A consistent database is tied to the notion of a transaction as an atomic operation, that is, an operation consisting of some defined processing activity bounded by an explicit beginning and an explicit end. The end itself is either an acknowledgement that the activity was successfully completed (commit) or an indication that it was not (abort), in which case the effects of the activity are undone by restoring the data to its original state (rollback).

While a transaction is in progress, the data that it modifies, though normally public, becomes private. Data that is modified by the transaction is considered volatile, meaning that it exists solely within the transaction. Only after the transaction is committed does the data become public again and the modifications made permanent. If the transaction is aborted, the effect on the data is as if no modifications had been made.

The processing activity depends on the application and potentially involves a series of events culminating in an update to the database. The transaction provides a useful means of combining physically separated actions into a single logical unit. In an airline reservation system, for example, booking a flight presupposes a prior determination that there is a flight to the destination with a seat available on a given date. Whatever the application, it is the user, not the system, who defines an atomic operation by bracketing a task with an explicit beginning and an explicit end.

Under Multics Data Management, consistency, concurrent access, locking, journalization, rollback, and similar integrity issues all have their basis in the context of a transaction as an atomic unit.

DEFINING TRANSACTIONS

A transaction is defined by the process that starts it, and so the process is said to be the owner of the transaction. A transaction can have only one owner process, and a process can have only one currently active transaction. A process must be in transaction mode (meaning it owns a currently active transaction) to access a protected DM file.

An application program executing in a user's process starts a transaction by calling transaction_manager_$begin_txn (see the description of the transaction_manager_ subroutine). Alternatively, a transaction can be started interactively through the transaction begin command (see the description of the "Command Level Interface to Data Management") which ultimately calls the same entry point.

The begin operation generates an entry in the transaction definition table (see below), in which the transaction is assigned a unique identifier and associated with its owner process. All database operations (i.e., MRDS dsl_ calls or LINUS requests) that comprise the task portion of the transaction are then executed in the user application program.

At the end of the processing activity, the owner process calls transaction_manager_$commit_txn (or the transaction commit command) to complete the atomic operation. Implicit in the commit operation is the release of the locks held by the transaction. If the processing activity cannot be successfully completed, the process calls transaction_manager_$abort_txn (or the transaction abort command) to roll back any database modifications and release the locks held by the transaction.

Other operations of interest to the caller of transaction_manager_ include:

suspend
: places the transaction in a suspended state during an interruption such as a quit signal or interactive message.

resume
: allows a suspended transaction to resume executing following an interruption.

rollback
: changes all modifications made by the transaction back to their original state and allows the transaction to be restarted from its beginning.

abandon
: relinquishes ownership and control of a transaction to a privileged daemon process (Data_Management.Daemon), which aborts it (see below).

kill
: bypasses the recovery mechanism when database consistency is inconsequential (e.g., if the database is to be deleted) to avoid the expense of recovery; this call requires privileged access to dm_daemon_gate_.

Note that both users and system modules call the transaction manager to invoke these operations. For a complete description of these and other calls to transaction_manager_, see the *Multics Subroutines and I/O Modules* manual, Order No. AG93.

BUILDING TRANSACTIONS IN EXISTING MRDS APPLICATIONS

Existing MRDS applications can be used with Data Management to obtain the benefits of integrity services. To do so, the database must be in DM file format (see "Using MRDS with Data Management"). The objective then is to take the applications programs that manipulate these databases and define them in terms of discrete tasks, consisting of one or a series of database calls. These tasks are considered atomic operations; they are defined as transactions by bracketing the database calls with calls to transaction_manager_$begin_txn and transaction_manager_$commit_txn or transaction_manager_$abort_txn as appropriate.

The following scenario is provided as a guideline for creating transactions within MRDS applications. One thing to remember with transactions is that rollback and abort operations affect only protected DM files. Other pieces of storage (e.g., static variables) may be relevant to the application and should be refreshed upon a rollback or abort to ensure restoration of the applications environment.

1. Identify whatever unprotected storage will need to be refreshed.

2. Define the dsl_ calls that comprise the atomic operation.

3. Check for an existing transaction in the process by calling transaction_manager_$get_current_txn_id. If the code dm_error_$no_current_transaction is returned, you can proceed.

4. Set up a cleanup handler to call transaction_manager_$abort_txn. As a safeguard in case the abort fails (i.e., a nonzero code is returned), it should call transaction_manager_$abandon_txn.

5. Set up an any_other handler to ascertain conditions signaled in the process (find_condition_info_). If transaction_deadlock_ or transaction_lock_timeout_ is signaled, the transaction has been rolled back automatically. Restart, refreshing unprotected storage as required. If transaction_bj_full_ is signaled, the transaction has been aborted automatically. If some other condition is signaled, call transaction_manager_$handle_conditions.

6. Call transaction_manager_$begin_txn and execute the dsl_ calls.

7. Examine the code returned by MRDS following the dsl_ calls. If the code is 0 or something other than an error, call transaction_manager_$commit_txn. Otherwise, you should decide whether to roll back and restart, or abort, remembering to refresh unprotected storage as required.

8. Revert the cleanup and any_other handlers.

The conditions identified in step 5 are described elsewhere in this section in the context in which they occur; also see Section 7. If a commit or rollback operation fails (i.e., a call to the respective entry point returns a nonzero code), you should call transaction_manager_$abort_txn. If the abort operation fails, you should call transaction_manager_$abandon_txn. It is also advisable to keep a count of the number of times a transaction is rolled back to avoid an infinite loop (if MRDS wrapped the transaction, it retries once before aborting on a deadlock condition).

Existing programs do not have to be modified to ensure the integrity of converted MRDS databases. In this case, virtually every MRDS operation is transformed into an atomic unit; i.e., each MRDS dsl_ call is automatically preceded by a call to transaction_manager_$begin_txn and followed by a call to transaction_manager_$commit_txn or transaction_manager$abort_txn, as appropriate. Thus, the user process can spawn a series of transactions by doing nothing more than executing an existing MRDS applications program that accesses a protected DM file. The tradeoff in not modifying programs is to create a transaction-intensive environment.

Transaction Definition Table

Initializing the data management system (see "DMS Initialization") creates an invocation of DMS. Information about transactions started during a DMS invocation is maintained in a table called the transaction definition table or TDT. The TDT contains an entry for each transaction; the entry includes the unique transaction id, the process id and Person_id.Project_id of the owner, date-time of the begin operation, current state of the transaction (e.g., in progress), and any available error information. The transaction manager is the overseer of the table, but the lock, before journal, and file managers all maintain a portion of each TDT entry.

Information available in the TDT can be retrieved by calling transaction_manager_$get_txn_info. In addition, the transaction status command can list selected information on any or all entries in the table, given the proper access (see "Command Level Interface to Data Management").

The TDT is created and initialized during DMS initialization. The first time any part of DMS is invoked by a user process, an entry is reserved in the TDT for the calling process. This entry is now available to the process when it wants to start a transaction. A call to the begin operation registers a transaction by generating a unique transaction id, binding the transaction to its owner process, and otherwise setting up the entry to track the transaction.

Most DM operations (e.g., $txn_begin) require a successfully initialized invocation of DMS to be available. If there is no successfully initialized invocation, or if no entry is available in the TDT for the current invocation of DMS, the dm_not_available_ condition is signaled in the process (see Section 7). The default action is to print a message and return to command level.

Transaction registration information is retained until the DM daemon (Data_Management.Daemon) determines that it is no longer needed and frees the entry for subsequent use. The TDT, as it pertains to recovery, is described below under "Process Failure" and "Crash Recovery."

Concurrent Access Control

To provide an optimal level of concurrency and still ensure database consistency, Multics Data Management synchronizes concurrent transaction activity by satisfying the serialization property, which requires that the result of concurrent transactions be equivalent to some serial execution of the transactions. This global property can be satisfied because all transactions observe the rules of concurrent access protocol as follows:

1. A transaction will never modify data modified by another transaction in progress.

2. A transaction will never read data modified by another transaction in progress.

3. A transaction will never modify data read by another transaction in progress.

The concept of locking (securing an object for the purpose of controlling access to that object) makes the concurrent access protocol enforceable. Locking is described in greater detail below.

LOCKING CONVENTIONS

A lock is associated with a piece of data and granted to a transaction for the purpose of controlling concurrent access to that data. The piece of data is said to be a lockable object, which defines the level of granularity within a system. Under Multics Data Management the control interval, as the unit of data transfer, is the lockable object by default.

The user has the flexibility to impose a coarser granularity (i.e., to lock the entire DM file) to improve performance. For example, when an activity requires that each tuple in a relation be examined for a particular value, it is more efficient to lock the entire relation at the outset than to lock the affected control interval as each tuple is retrieved. This is known as lock advice, to be set as a file attribute when the file is first opened (see the description of file_manager_). Lock advice is just that, an advisory, so that a file would never be placed in jeopardy because of it. DMS is always the final arbiter in deciding what constitutes sufficient locking to guarantee file consistency.

A transaction that modifies a piece of data (resulting in a put call to the file manager) acquires a write lock on the containing control interval(s). A write lock is granted only to one transaction and is said to be exclusive. The file manager, upon receiving the put call, verifies that the process is in transaction mode, then calls the lock manager to set the proper locks on behalf of the transaction.

A transaction that retrieves a piece of data (resulting in a get call to the file manager) acquires a read lock on the containing control interval(s). A read lock is granted to one or more transactions and is said to be shared. The file manager, upon receiving the get call, verifies that the process is in transaction mode, then calls the lock manager to set the proper locks on behalf of the transaction.

The rules of locking are imposed automatically and can be stated as follows:

- Every transaction must acquire a read lock to read an object and a write lock to modify an object.
- All locks acquired by a transaction are held by that transaction until it is committed or aborted.

To avoid conflicts in locking (and having to investigate for potential conflicts each time a lock is requested), the share and exclusive lock modes on a control interval are supplemented by additional lock modes on the entire file known as intention locks. For example, a transaction that acquires a share lock on a control interval implicitly acquires an intended share lock on the containing file, so that another transaction cannot lock the file in exclusive mode.

Calls to the lock manager are timed so that transactions will not wait indefinitely to acquire locks. If the wait-time elapses before a lock can be set, DMS automatically rolls back the transaction and signals the transaction_lock_timeout_ condition in the process (see Section 7). The default action is to print a message and return to command level.

Locking can be turned off at file creation for a given file by turning the no_concurrency bit on in the dm_file_create info structure (the bit setting is off by default). If protection from concurrent access is irrelevant, this option can be selected to avoid the overhead of locking.

DEADLOCK DETECTION AND RESOLUTION

Whenever there is concurrent access, there is potential for the so-called deadly embrace, whereby two or more transactions are stalled, waiting on each other to release the locks on data each seeks to access. None of the transactions involved can resume processing until one of them frees its data, resolving the deadlock.

The lock manager is charged with deadlock detection. If a request for a lock, if granted, would result in a deadlock, the lock manager selects the "youngest" transaction involved to be rolled back, freeing the data so that the other transactions can resume processing. The transaction_deadlock_ condition is then signaled in the process owning the youngest transaction (see Section 7); it can either abort or retry the transaction.

If your transaction was wrapped automatically by MRDS, a retry is attempted after the automatic rollback, and if unsuccessful, the transaction is aborted and the dm_error_$lock_deadlock code is returned. Transactions started by the transaction execute command have built-in condition handlers.

The lock manager provides further deadlock detection by examining all requests for locks, tracking those transactions forced to wait because a lock cannot be granted. When the wait is caused by another transaction holding a lock on the data, and the wait exceeds a predefined threshold, the suspicion is that the owner process of the holding transaction no longer exists. If its owner process is "dead," the transaction cannot finish, and there is a deadlock condition.

The lock manager calls upon the DM daemon to investigate the status of the owner process and the daemon, finding that the process no longer exists, resolves the condition by aborting the unfinished transaction and releasing the locks it held so the waiting transaction can resume. The daemon's actions in this regard are described below in greater detail under "Process Failure" and "Role of the Daemon."

Recovery Procedures

Recovery refers to the restoration of the database, in this case protected DM files, following some type of failure which may have threatened database consistency. The recovery procedures consist of rolling back database modifications made by transactions that were not properly terminated, and releasing the locks held by these transactions. Recovery is possible because images of the data before it is modified are preserved in a before journal, available to restore the database to its original state in the event of failure. Before journals are described in detail below under "Conventions and Use of Before Journals."

The type of failure dictates the environment in which recovery takes place. If the transaction itself fails, recovery is accomplished within the owner process in the form of a call to transaction_manager_. If the owner process fails (becoming a dead process), the DM daemon assumes responsiblity for recovery. In the event of system crash, the DM daemon activates recovery procedures following reinitialization of the system. Each type of failure is discussed below in terms of how recovery is implemented. A note of caution on recovery: if the directory containing the DM file to be rolled back is renamed, the daemon will be unable to locate the file on rollback (see "Manipulating Data Management Files").

There is also the possibility of media failure, involving physical damage to the disk. Systematic volume dumping affords the best method of protection against this type of failure. It is important that dumping be performed at a time when no updating of DM files is taking place.

TRANSACTION FAILURE

Transaction failure can be triggered by any call that returns a nonzero error code in the process, or by releasing on a sub_error_ condition, or, more often, on some subjective evaluation by the user. When a transaction is deemed a failure, the owner process (user application program) must abort the transaction by calling transaction_manager_$abort_txn. The effect of the abort operation is to "undo" any modifications caused by the failed transaction and to release all locks held by it. All references to the transaction are removed from the appropriate DMS system tables so it appears as if the transaction had never existed.

Alternatively, the owner process can call the rollback operation (transaction_manager_$rollback) to effect a rollback to the beginning of the transaction, which may then be restarted. In this case, the transaction retains its id and entry in the TDT.

The recovery mechanism set in motion by a call to transaction_manager_$abort_txn involves several internal calls to other integrity services modules. Just as with data storage and retrieval, there is a modular breakdown of responsibilities, ensuring that the same software performs the same function in the same way all the time. The transaction manager calls the before journal manager to perform the rollback. The before journal manager locates the before images for the given transaction, but the file manager actually adjusts the DM file. When the database has been restored, the transaction manager calls the lock manager to release the locks held on behalf of the aborted transaction. At the completion of these activities, the database is again consistent; the data made private by the transaction becomes public; and, the process is free to start another transaction.

PROCESS FAILURE

A process that fails, on a stack overflow or hangup condition for example, may leave a transaction unfinished. The event in itself does not threaten database consistency because the data remains locked, but the transaction must be aborted to roll back any modifications and free the data for other transactions. Since the process is dead, the DM daemon must assume responsibility for terminating the unfinished transaction.

In the TDT, the process id is associated with the transaction id assigned when the transaction entry was created. The DM daemon scans the list of process ids and checks with the ring zero supervisor to verify the state of each process. If a process no longer exists, the daemon then checks the state of the associated transaction. If the check reveals that the transaction has been committed or aborted, the daemon simply removes the transaction's registration information from the table. For a transaction listed as in progress, however, the daemon calls transaction_manager_$abort_txn, setting in motion the chain of events described for transaction failures. The daemon then arranges for removal of the TDT entry, as above.

Role of the Daemon

The DM daemon has specific responsibilities with regard to DMS initialization, shutdown, and crash recovery scenarios. These responsibilities are described under the appropriate headings below. Within an active DMS invocation, the daemon's sole function is to abort unfinished transactions owned by dead processes and to remove transaction registration information from DMS tables. It performs this "caretaker" function by scanning the TDT at regular intervals (an idle timeout alarm), or upon direct notification from the answering service that a dead process exists.

It is possible for other processes to discover dead processes. So it might appear that they should also be able to abort the unfinished transactions of these processes in the interest of keeping the database consistent and releasing the locks held by these unfinished transactions.

Aborting a transaction involves writing in DM files, however, which implies access to these files, and to force access constitutes a violation of access principles. To uphold these principles, only the DM daemon can abort a transaction left unfinished by a process that no longer exists (or has relinquished control through an abandon operation; see below). Thus, a process that discovers a dead process sends a wakeup message to the daemon, which verifies then proceeds with its caretaker function.

When the lock manager encounters a deadlock situation, it sends its own wakeup message to the daemon to investigate. The message includes the process id of the owner of the transaction that forced the deadlock. The daemon verifies that the process no longer exists and aborts the unfinished transaction (i.e., calls transaction_manager_$abort_txn), permitting the deadlocked transaction to proceed.

Abandoning a Transaction

The abandon operation (transaction_manager_$abandon_txn) provides an escape mechanism by which a user process can disown a transaction without properly terminating it. Having relinquished control of the one transaction to the DM daemon, the user process is free to start another. The daemon "adopts" the abandoned transaction to roll back any modifications and release whatever locks it held (effectively aborting it).

A process that abandons a transaction cannot regain control of it. If the process starts another transaction and attempts to access the same data, it may have to wait until the daemon releases the locks held by the original transaction.

A user process can also kill a transaction (transaction_manager_$kill_txn), an operation which bypasses the recovery mechanism altogether. This call requires access to dm_daemon_gate_, and should be used only when database consistency is inconsequential, for there is no avenue of recovery once the kill operation is executed.

CRASH RECOVERY

Protected DM files are guaranteed recoverable in the event of a system crash of DMS or of Multics itself. Recovery is possible because certain tables in the DMS bootload directory (as described under "DMS Initialization" below) are guaranteed reliable in the aftermath of the crash. These tables contain information relevant to all DM files and all before journals that were open at the time of the crash. Each time during a DMS invocation a DM file or a before journal is opened and registered in the respective table, the table is written out to disk, ensuring its reliability.

Reliability is assured whether or not emergency shutdown (ESD) is successful, because of the write ahead log (WAL) protocol, which guarantees that no physical modification of the database occurs until its before image is preserved on disk. The WAL protocol is enforced through the cooperation of page control, as the manager of Multics virtual memory, and the file and before journal managers as overseers of protected DM files. Pages containing modified control intervals are said to be "sync-held" until the corresponding before images are verifiably flushed to disk. If a loss of main memory occurs (i.e., there is ESD failure), the sequence of migration to disk ensures that no vital recovery data is part of that loss.

If the system fails, because of a power failure for example, recovery occurs following system reinitialization, when the daemon process is logged in (see "DMS Initialization"). The charter of the daemon at startup is to determine if recovery is necessary, and if so, to identify those transactions that were active at the time of the crash and abort them. Then the daemon enables the current DMS invocation so that new transactions can be started.

If a normal shutdown is indicated (see "DMS Shutdown"), recovery is unnecessary. The daemon enables the current DMS invocation and assumes its caretaker function.

When the daemon discovers at startup that recovery is necessary, it must in effect re-create the environment of the previous invocation in order to abort the unfinished transactions and make the database consistent. It does this by calling the various integrity services modules to perform the following functions:

- Open all before journals open at the time of the crash (this information is available in the table of open journals preserved across a system crash)
- Build a temporary TDT of all unfinished transactions from the previous invocation (accomplished by reading each before journal in reverse chronological order; see "Conventions and Use of Before Journals" below)

- Open all protected DM files that may have been modified during the previous invocation (this information is available in the table of unique file ids and pathnames, i.e., all open DM files, preserved across a system crash)

The daemon then aborts each unfinished transaction listed in the temporary TDT, just as it would after discovering a dead process or adopting an abandoned transaction. DM files are thus restored to the state they were in prior to any update activity caused by the unfinished transactions during the previous DMS invocation.

When recovery procedures are completed, the daemon arranges for the before journals and DM files to be closed, disposes of the old tables used for recovery, and enables the current DMS invocation so that new transactions can be started.

CONVENTIONS AND USE OF BEFORE JOURNALS

A before journal is a type of DM file, written as a sequential disk file of variable-length logical records, physically grouped as control intervals. These records are written in the order in which they are produced by transactions using the journal. While there may be multiple journals open within a DMS invocation or even within a process, the before images generated by a given transaction must all be recorded in the same before journal.

The caller of transaction_manager_$begin_txn can specify which before journal to use; otherwise, a default journal is assigned (see "Creating and Opening Journals," below). The begin operation generates a call from the transaction manager to the before journal manager to write a begin mark in the designated journal. It is this mark that delineates the rollback operation for an aborted transaction.

When a transaction modifies a control interval in a protected DM file, the file manager calls the before journal manager to record the before image in the proper journal. All before images recorded for a given transaction are chained together to facilitate rollback, if necessary.

If the transaction is terminated, the before journal manager writes the appropriate mark (either commit or abort) at the direction of the transaction manager to denote the transaction as finished. Before images for committed and aborted transactions are no longer of any use and can be overwritten.

Even though space in a journal is reuseable, the journal may appear to be full if the daemon has not yet adjusted terminated transactions. When a transaction attempts to write a before image to a full journal, the transaction is aborted automatically and the transaction_bj_full_ condition is signaled in the process (see Section 7). The default action is to print a message and return to command level. The same sequence of events occurs if the transaction exceeds its storage limits on the journal when attempting to write a before image (see "Creating and Opening Before Journals" below).

In cases where MRDS has wrapped the transaction automatically or if you have used the transaction execute command, cleanup handlers are set up to handle a full condition. If you release, you unwind past the cleanup handler and the transaction is aborted as above.

Journalization can be turned off at file creation for a given DM file by turning the no_rollback bit on in the dm_file_create info structure (the bit setting is off by default). If file integrity is irrelevant, this option can be selected to avoid the overhead of journalization.

During crash recovery, each journal open in the previous invocation is positioned to the last control interval and read in reverse chronological order to identify all transactions in progress at the time of the crash. A time stamp convention is used to locate the end of the file. Since the file is a closed loop and control intervals are written sequentially within a DMS invocation, the beginning of the file is reached when the time recorded in the control interval header stops increasing. Page control also uses the time stamp to verify whether a before image has been written to disk before flushing the page containing the DM file modification to disk (see the discussion of the WAL protocol under "Crash Recovery").

The opened journals are processed serially. If no unfinished transactions are found in a journal, it is closed and the next one read until all journals have been processed, or until the expected number of unfinished transactions has been identified. The daemon knows how many transactions have to be recovered because each before image is recorded with a count of the number of active transactions using that journal. The last before image logged before the crash contains the number of transactions then in progress and, hence, the number of transactions that have to be accounted for in a given journal.

Creating and Opening Before Journals

Before journals are created by calling before_journal_manager_$create_bj (see the before_journal_manager_ subroutine description) or through the bj_mgr_call create command (see the description of "Command Level Interface to Data Management"). In either case, the size of a journal is defined as a number of control intervals and would be based on evaluation and analysis of metering information concerning the number of transactions to be using the journal, their average time, and rate of modification. Currently, the default journal size is set at 64 (4096-byte) control intervals.

Part of before journal creation is to limit the amount of space any one transaction can demand in recording its before images. The objective in setting limits on individual transactions is to enhance overall performance. If no limit is set, a transaction can theoretically commandeer the entire before journal, forcing other transactions attempting to write to the same journal to abort.

Journals must be opened in the process in which they are to be used. Multiple journal openings are permitted within a process, but a journal must be closed the same number of times it is opened. A single close operation merely decreases the number of openings by one. Journals are opened and closed in a process by calling the respective entry point in the before_journal_manager_ subroutine or by using the bj_mgr_call command.

Transactions started in a process without specifically naming a before journal are assigned one according to a default scheme. A default journal for the process can be explicitly set by calling before_journal_manager_$set_default_bj. The journal so designated must be open in the process. If no such journal is designated, the journal most recently opened in the process is used. If no journal is currently open in the process, the system default before journal is automatically opened in the process and assigned to the transaction being started.

Manipulating Before Journals in the File System

Before journals are considered extended entry types and as such they can be manipulated in the Multics storage system through many of the standard commands by including the .bj suffix as part of the entry name. The file system utility provides a suffix handler for the .bj suffix, so no user-written routine is required (see the description of fs_util_ in the Subroutines manual). The following file system commands recognize before journals as extended entry types:

```
add_name
copy_names
delete
delete_acl
entries
exists
list_acl
set_acl
status
```

DMS Initialization

Data Management System initialization is part of the Multics bootload operation and results in an invocation of DMS. The run-time DMS comprises a set of per-system tables, created and initialized as segments in a DMS bootload directory at the time of the Multics bootload. Since tables from the previous invocation are integral to crash recovery (see the description of "Crash Recovery"), the per-bootload DMS directory is saved from one bootload to the next, unless a successful shutdown is indicated (see the description of "DMS Shutdown"). Multiple per-bootload directories can exist (usually for debugging purposes); they are distinguished from one another by the time of the Multics bootload.

When Multics is booted, Data_Management.Daemon, like other daemon processes, is logged in either by the system_start_up.ec or by explicit operator command. The daemon's initial responsibility is to bootload DMS by calling the data management initializer program, a set of ordered calls to specialized initialization and recovery routines. The daemon's subsequent responsibilities with regard to initialization can be summarized as follows:

- Locate the DMS system tables from the previous invocation to see if recovery is necessary. These tables may not exist if the previous invocation was successfully shutdown or if there was no previous invocation.
- Build new system tables for the current invocation, but prevent their use temporarily. DMS is not available until recovery is completed.
- Perform recovery if necessary (see the description of "Crash Recovery").
- Enable the new system tables for activity under this DMS invocation.
- Delete (or save for debugging purposes) the system tables from the previous invocation.

If the daemon cannot locate the previous DMS bootload directory, no recovery is necessary. This is an indication that the previous invocation was shut down normally, and that the daemon deleted the old DMS directory prior to logging out. Initialization of the current system invocation then proceeds with the building of per-system and tables.

If recovery is necessary, that part of DMS initialization proceeds as described under "Crash Recovery." When recovery is completed, the new system tables are enabled for activity under this invocation, and the old directory is disposed of or saved, as appropriate.

Occasionally the daemon may discover that DMS is currently running. The implication is that the old daemon process died, leaving the system unattended. In this case, the new daemon merely has to take over the running system, cleaning up unfinished transactions as it normally would performing the caretaker function.

If your site runs with the Access Isolation Mechanism (AIM), a directory must be created for each AIM classification that is to use Data Management. This directory is referred to as the per-AIM directory and is created under the per-system directory, >site>Data_Management. Everything described for a DMS invocation at Multics bootload occurs for each AIM classification using Data Management.

DMS Shutdown

DMS has its own shutdown procedures, both as added assurance that protected DM files are left in a consistent state and to reduce recovery time at the next bootload. Crash recovery is an integral part of DMS initialization, during which no users are allowed into the DMS environment. If, at the start of initialization, crash recovery can be ruled out, the system becomes available almost immediately.

If you attempt to access any part of DMS during those periods of initialization or shutdown when the system is inaccessible, the dm_not_available_ condition is signaled in your process (see Section 7). The default is to print a message and return to command level. The same condition is signaled if no invocation of DMS exists.

DMS shutdown is designed to leave no transactions in progress and all protected DM files and before journals closed (and therefore consistent) when the DM daemon is logged out. Shutdown occurs either as part of a Multics shutdown or by privileged intervention of the data management administrator or operator (see the dm_system_shutdown command described in the *Administration, Maintenance, and Operations Commands* manual, Order No. GB64).

DMS SHUTDOWN AS PART OF A MULTICS SHUTDOWN

When a Multics shutdown is scheduled, the system_shutdown_scheduled_ interprocess signal (IPS) triggers the DM daemon to schedule a DMS shutdown. The daemon records the pertinent shutdown information in dm_system_data_ (a per-system table in the DMS directory) and sends the dm_shutdown_scheduled_ IPS to notify all DMS users of the scheduled DM shutdown. If a Multics shutdown is already scheduled when DMS is enabled following per-system initialization, the daemon proceeds with the scheduling of the DMS shutdown as described above, without the system shutdown notification.

Subsequent steps to DMS shutdown are as follows:

1. By default, two alarm call channels are set up for each affected user process, one for "user warning" and one for "user shutdown." These alarm calls are set up when the dm_shutdown_scheduled_ IPS is received or as part of per-process initialization, if the process had not yet invoked DMS when the IPS was sent.

2. When the user warning alarm goes off, a dm_shutdown_warning_ condition is signaled in the user process, and a message announces that shutdown is pending.

3. When the user shutdown alarm goes off, a dm_user_shutdown_ condition is signaled in the user process, and a message announces that user shutdown is underway.

4. The DM daemon then requests the answering service to bump any still-active DMS users (in the same manner that users can be bumped from Multics).

5. The daemon logs itself out when normal shutdown occurs, or when the scheduled DM daemon logout time is reached.

Each of these events is described below.

DMS SHUTDOWN AS A PRIVILEGED REQUEST

The data management administrator or the system operator can shut down DMS, irrespective of a Multics shutdown, by invoking the dm_system_shutdown command as described in the *Administration, Maintenance, and Operations Commands* manual. In this case, the command triggers the daemon to send the dm_shutdown_scheduled_ IPS to all DMS users; the remaining shutdown steps are as described above. Any differences in the implementation of a given step are noted below in the description of that step.

The requestor must have re access to dm_admin_gate_ to execute this command.

SHUTDOWN INFORMATION

Information regarding DM shutdown is recorded in dm_system_data_, a segment in the DMS bootload directory, either as part of per-system initialization or as a result of the dm_system_shutdown command, and includes the following:

- Reason for shutdown (either because of a scheduled Multics shutdown or as stated by the administrator or operator)
- User warning time
- Begin shutdown time
- User shutdown time
- User bump time
- Daemon logout time

This information, excluding user bump and daemon logout times, is available for user inspection.

Unless otherwise specified, user bump time coincides with the scheduled Multics shutdown time. All other times are calculated backward or forward according to specified or default time delays contained in the DMS configuration file (see "Administering Data Management"). The command can explicitly specify all times and all delays, or any combination. The system default delay time between events is five minutes.

Also tracked in dm_system_data_ is the state of DMS; this value progresses from "running" to "begin shutdown," "user bump," and "normal shutdown" as the DMS shutdown itself proceeds. In cases where the DM daemon dies, this status indicator informs the new daemon where to resume the shutdown operation.

User Warning

User warning time provides a grace period in which to conclude transaction activity and close all protected files and before journals open in the process.

When the user warning alarm goes off, the dm_shutdown_warning_ condition is signaled in the process (see Section 7). Unless the condition is handled, a message denotes when no new transactions will be accepted ("begin shutdown" time), when a forced shutdown will occur ("user shutdown" time), and the reason for the shutdown. The process then resumes executing at the point of interruption. All transactions should be finished, and all files closed, in this time frame.

For absentee processes, no message is printed on output at user warning time, regardless of whether the default handler is overridden, so as not to disrupt any special formatting of output.

If a user process starts using DMS after the warning time has passed, but before new transactions are prevented ("begin shutdown" time), the dm_shutdown_warning_ condition is signaled as part of per-process initialization.

Begin Shutdown

After the grace period has elapsed, the DM daemon changes the state of DMS to "begin shutdown." No new transactions are permitted to be started under this invocation of DMS, but currently active transactions are allowed to continue.

User Shutdown

At user shutdown time, any transactions still active (but not attempting to finish) are aborted or forcibly abandoned so that the DM daemon can abort them, using normal rollback procedures. Additionally, all user process references to per-process and per-system data are invalidated, to enable reentry to DMS on a subsequent DMS bootload.

When the user shutdown alarm goes off, the dm_user_shutdown_ condition is signaled in the process (see Section 7). The default action is an automatic call to transaction_manager_$user_shutdown to terminate DMS in the process by adjusting any TDT entry belonging to that process. A message announces that user shutdown is underway, and the process resumes executing at the point of interruption.

User Bump Time

Bump time occurs after user shutdown, following a specified delay. When bump time is reached, the daemon scans the TDT, searching for any remaining DMS users, and asks the answering service to bump those that are found. For a DMS shutdown that is part of a Multics shutdown, this event coincides with the system shutting down, when all users remaining on the system are bumped.

Daemon Logout

The daemon logs out when shutdown is complete or when the scheduled logout time is reached. Shutdown can actually be complete any time after the "begin shutdown" step. Such is the case when all TDT entries have been adjusted, and the daemon is the only user remaining. If this is so, the daemon sets the state to "normal shutdown" and arranges for disposition of the DMS directory (as would be done following crash recovery, if performed as part of initialization). The daemon then logs itself out without waiting for any of the other scheduled events to occur.

ADMINISTERING DATA MANAGEMENT

As with all aspects of Multics, the system administrator has specific responsibilities with regard to preparing and controlling the environment in which Data Management operates. These responsibilities are outlined below. Details are provided in the section entitled "Data Management Administration" in the *System Administration Procedures* manual, Order No. AK50.

Installation Considerations

As part of Multics installation, you must include a dbmj card in the config deck to configure before journal management. Additionally, you must provide for directory flushing by switching on the dirlock_writebehind tuning parameter either by including the parm dirw card in the config deck or by using the change_tuning_parameters command (see the *Multics System Maintenance Procedures* manual, Order No. AM81, for a complete description of the config deck).

Creating a Data Management System Directory

You must create a data management system directory (the default location is >site>Data_Management). This directory will contain all the per-AIM directories (see "AIM Considerations" below). If you are not running with AIM, you must create the directory system_low as the per-AIM directory for level zero and no categories. The contents of the per-AIM directory include the data management configuration table (see "Shaping the Run-Time Environment"), the system default before journal, the logs directory, and the DMS per-bootload directory created automatically at DMS initialization.

Shaping the Run-Time Environment

You must create a configuration source file and convert it into a data management configuration table using the cv_dmcf command (see the *Administration, Maintenance, and Operations Commands* manual, Order No. GB64). You must then install the table in the per-AIM directory, where it is used to establish the run-time parameters of DMS, as follows:

- Size and location of the system default before journal
- Maximum number of processes that can use DMS
- Maximum number of transactions active at any time
- Idle timeout, i.e., automatic daemon wakeup to adjust dead processes
- Begin shutdown delay (delay between user warning and begin shutdown)
- User shutdown delay (delay between begin shutdown and user shutdown)
- User bump delay (delay between user shutdown and user bump)
- Daemon logout delay (delay between user bump and daemon logout)
- Previous bootload disposition (delete or retain for debugging purposes)
- Log process terminations (an instruction for the daemon)
- Enable the current bootload despite inability to recover from previous bootload (turned on only on instruction from Multics Development Center)

Daemon Registration

The DM daemon (Data_Management.Daemon) is required to run Data Management on Multics. It serves as caretaker during DMS operations by recovering abandoned transactions and transactions belonging to dead processes. It starts DMS initialization after a Multics bootload and performs crash recovery. It is also the daemon that schedules and controls data management shutdown.

Data_Management.Daemon, as part of the Daemon project, is a special user identity, automatically registered and installed when Multics is initialized at a new site. It has its own process overseer (dmsd_overseer_), established by the initproc attribute in its PMF, which handles all conditions and signals to the daemon, sets up an event channel for requests to the daemon, and registers itself with the answering service to be notified of process terminations.

A DM daemon is required for each AIM classification using Data Management (see below).

AIM Considerations

You must create a per-AIM directory under the DMS system directory for each AIM classification using Data Management. You must also provide a data management configuration table and DM daemon for each AIM classification.

Monitoring Performance

Various privileged commands (see "Command Level Interface to Data Management") are available that meter system activity, providing a barometer with which to evaluate DMS performance. Evaluation and analysis will enable you to fine-tune different areas of the system to improve performance.

Additionally, separate data management logs are maintained as part of the system logging facility. Information regarding initialization, shutdown, recovery, daemon adjustments to the TDT, and various error exceptions are logged for each per-AIM directory using Data Management. This information can be viewed by using the print_sys_log and monitor_sys_log commands with -dm_system control argument (see the *Administration, Maintenance, and Operations Commands* manual, Order No. GB64).

COMMAND LEVEL INTERFACE TO DATA MANAGEMENT

Multics Data Management supports a command level interface consisting of two sets of commands, one set for users to invoke integrity services interactively, and the other for administrators to set up, control, and monitor the DMS environment. Both sets of commands are described below.

User Commands

Listed below are the DMS commands available to all users. Each command is summarized following the list; for complete descriptions, see the *Multics Commands and Active Functions* manual, Order No. AG92.

```
transaction
bj_mgr_call
before_journal_status
create_dm_file
dm_user_shutdown
dm_display_version
```

The transaction command provides users with the ability to define and run transactions interactively during a terminal session. The following operations are possible: begin, commit, abort, execute, rollback, abandon, and kill. There is also a status operation with which to select varied information about your current transaction, or all transactions, if you have the proper access.

Operating from command level, you can start a transaction in your process, perform the desired database operations, and consider the results before committing the transaction. You are actually writing your application program in real time. You can also react to error conditions as they occur. This method is best suited for testing.

The transaction execute command is intended for running production applications from command level. The execute operation allows you to wrap an application in a transaction within a single command line. This operation has built-in condition handlers, setup to catch the conditions signaled by Data Management.

The bj_mgr_call command provides users with the ability to create and manipulate before journals in the user process. You can create, open, and close before journals, set and get the default journal for the process, and ascertain whether specified journals are already opened or closed.

The before_journal_status command provides information on specified journals within the user process, or all journals active within DMS, given the proper access. The information displayed includes journal identifier, size, whether in use, and the times at which before images are buffered and written.

The create_dm_file command enables you to create unpopulated DM files, with or without protection, from command level. You must have sm access on the containing directory if you are creating a protected DM file. You would normally use this command in cases where you are using file_manager_ as a direct interface.

The dm_user_shutdown command enables you to remove your process from active status in the current invocation of DMS. All user process references to per-process and per-system data are invalidated to allow subsequent reentry to DMS. If there is a transaction in progress, it is aborted, unless it is committing, in which case it is allowed to finish. If the transaction cannot be aborted for any reason, it is abandoned.

The dm_display_version command enables you to ascertain which version of DMS you are currently using.

Administrative Commands

Listed below are the administrative commands available to users with access to dm_admin_gate_ or which, by the nature of their function, are considered privileged. Each command is summarized following the list; complete descriptions of these commands appear in the *Administration, Maintenance, and Operations Commands* manual, Order No. GB64.

```
cv_dmcf
dm_system_shutdown
dm_lock_status
dm_lock_meters
before_journal_meters
dm_set_journal_stamps
dm_set_system_dir
dm_send_request
```

Administrators use the cv_dmcf command to convert the configuration source file into a data management configuration table to be used in initializing DMS. The configuration table contains the system-default parameters for the running DMS. It must be installed in the per-AIM directory.

Administrators and operators use the dm_system_shutdown command to shut down DMS while Multics is still running. The shutdown consists of specific stages in which users are warned of the impending shutdown, new transactions are prevented from starting, user processes are shut down (active transactions are allowed to finish if committing; otherwise they are aborted or forcibly abandoned), remaining users are bumped from DMS, and the daemon logs out.

Administrators use the dm_lock_status command to view all locks currently held or awaited by all transactions.

Administrators use the dm_lock_meters command to examine metering information kept on all locking activity during a given period.

Administrators use the before_journal_meters command to examine metering information on journal use during a given period on a system and per-journal basis. Non-privileged use of this command provides metering information on the use of those journals to which the requestor has access.

Administrators use the dm_set_journal_stamps command to release all sync-held pages by setting the time stamp in all journals currently in use to the time that the command is invoked. This in turn will cause all sync-held pages to be flushed to disk, which may endanger recovery in case of a crash without ESD.

It may be necessary to invoke this command if before journals become damaged, preventing transactions from being committed, or if there is an unusually large number of sync-held pages, stalling system performance. This command provides a means of keeping the system running instead of shutting it down. It should be used judiciously.

Administrators use the dm_set_system_dir command to set up a data management per-system directory in their own process, for test purposes. For normal processing, the DMS per-system directory at >site>Data_Management is used.

Administrators use the dm_send_request command under unusual circumstances to request certain actions of the daemon. Actions that can be requested include: kill a transaction, adjust a TDT entry, recalculate shutdown times, or do a new_proc.

APPENDIX A

MULTICS CHARACTER SETS

The Multics system uses the full ASCII character set for most languages. The Multics Extended Character Set, discussed below, is also available on a limited scale.

ASCII CHARACTER SET

Multics uses the revised ASCII Standard (refer to American National Standards Institute, "ANSI Standard X3.4-1968"). The set consists of 128 7-bit characters, including 94 printing graphics, 33 control characters, and the space.

Printing Graphic Characters

The ASCII printing graphic characters are the uppercase alphabet, the lowercase alphabet, digits, and a set of special characters (see Table A-1). The special characters are listed below.

```
! exclamation point          ; semicolon
" double quote               < less than
# number sign                = equals
$ dollar sign                > greater than
% percent                    ? question mark
& ampersand                  @ commercial at
' acute accent               [ left bracket
( left parenthesis           \ left slant
) right parenthesis          ] right bracket
* asterisk                   ^ circumflex
+ plus                       _ underline
, comma                      ` grave accent
- minus                      { left brace
. period                     | vertical line
/ right slant                } right brace
: colon                      ~ tilde
```

Control Characters

The following conventions define the standard meaning of the ASCII control characters that are given precise interpretations in Multics. These conventions are followed by all standard I/O modules and by all system software inside the I/O system interface. Since some devices have different interpretations for some characters, it is the responsibility of the appropriate I/O module to perform the necessary translations.

Table A-1. ASCII Character Set on Multics

	0	1	2	3	4	5	6	7
000	(NUL)							BEL
010	BS	HT	NL	VT	NP	CR	RRS	BRS
020								
030								
040	Space	!	"	#	$	%	&	'
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	`	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	}	~	PAD

The Multics standard control characters are:

BEL
Sound an audible alarm.

BS (Backspace)
Move the carriage back one space. The backspace character implies overstrike rather than erase.

HT (Horizontal tab)
Move the carriage to the next horizontal tab stop. Multics standard tab stops are at 11, 21, 31... when the first column is numbered 1.

NL (Newline)
Move the carriage to the left end of the next line. This implies a carriage return plus a line feed. ASCII LF (octal 012) is used for this character.

VT (Vertical tab)
Move the carriage to the next vertical tab stop and to the left of the page. Standard tab stops are at lines 11, 21, 31... when the first line is numbered 1.

NP (new page)
Move the carriage to the top of the next page and to the left of the line. ASCII FF (octal 014) is used for this character.

CR (Carriage return)
Move the carriage to the left of the current line. This character cannot appear in a canonical string.

RRS (Red ribbon shift)
ASCII SO (octal 016) is used for this character.

BRD (Black ribbon shift)
ASCII SI (octal 017) is used for this character.

PAD (Padding character)
This is used to fill out words that contain fewer than four characters and that are not accompanied by character counts. This character is discarded when encountered in an output line. It may appear in a canonical character string if the input/output conversion of a user's terminal is set properly. ASCII DEL (octal 177) is used for this character.

The characters designated as unused are specifically reserved and can be assigned definitions at any time. Until defined, unused control characters are written using the octal escape convention in normal output and are not printed in edited mode. Users wishing to assign interpretations for an unused character must use a nonstandard I/O module.

If a device does not perform a function implied by a control character, its standard I/O module provides a reasonable interpretation for the character on output. This might be substituting one or more characters for the character in question, printing an octal escape, or ignoring it.

Nonstandard Control Character

One control character, NUL, is recognized under certain conditions by all I/O modules because of its wide use outside Multics. This character is handled specially only when the I/O module is printing in edited mode, and is, therefore, ignoring unavailable control functions. The null character is ASCII character NUL (octal 000). In normal mode, this character is printed with an octal escape sequence; in edited mode, it is treated like PAD. This character may appear in a canonical character string if the input/output conversion of a user's terminal is set properly. Programmers are warned against using NUL as a routine padding character and using edited mode on output because all strings of zeros, including mistakenly uninitialized strings, are discarded.

Unused Characters

These characters are reserved for future use:

SOH	001	ACK	006	DC4	024	EM	031
STX	002	DLE	020	NAK	025	SUB	032
ETX	003	DC1	021	SYN	026	FS	034
EOT	004	DC2	022	ETB	027	GS	035
ENQ	005	DC3	023	CAN	030	RS	036
						US	037

MULTICS EXTENDED CHARACTER SET

The Multics Extended Character Set consists of 512 9-bit characters stored one per byte. It contains the ASCII character set as its first 128 characters.

Most Multics commands and subroutines use only the 128 ASCII characters. Multics APL, however, uses many non-ASCII characters to represent the extra graphics in its character set. The APL use of the Multics Extended Character Set is not a Multics standard; there is no standard for mapping non-ASCII character codes to graphic characters.

APPENDIX B

DEFINING TERMINALS AND NAMING CHANNELS WITHIN THE MULTICS COMMUNICATIONS SYSTEM

The Multics Communication System (MCS) effects the transfer of data between the Multics virtual memory and various remote devices (primarily terminals) over communications channels. *

The bulk of MCS resides in the Multics supervisor and in a separate machine, the Front-End Network Processor (FNP). The user-ring and supervisor portions of MCS are principally concerned with terminal management, while the FNP's primary responsibility is channel management. In general, the user need not be concerned with channel management. Most user and system programs interface to MCS through the input/output system by means of the iox_ subroutine, described in the Subroutines manual. For general information on the use of the I/O system, see Section 5.

TERMINALS AND CHANNELS

The term "channel" (or "communications channel"), as used here, refers to a physical connection between an FNP and a remote input/output device. Such a connection may go through a telephone system or a private communications network, or it may consist of one or more hardwired cables. For information on the specification and management of all communications channels known to the system, see the MAM Communications. The naming of channels is described later in this appendix.

The word "terminal" is used to refer to the device itself; it may be an ordinary interactive terminal on which a user types commands, or it may be a computer controlling one or more peripheral devices. The setting and changing of terminal types are described later in this appendix.

ATTACHMENTS

An interactive terminal is normally connected to the system (attached) through the tty_ I/O module (see the Subroutines manual). For the user's login terminal, this attachment is performed automatically in the course of process creation. Users who desire extended support for special video terminal features must attach their terminal to the "video system" via the window_io_ I/O module.

Other types of devices that use special communications protocols may have to be attached through special-purpose I/O modules. Several such modules are supplied with the system; they are described in the Subroutines manual. Users and sites may also supply their own I/O modules that interface to one of the existing modules (see Section 5).

Additional terminals can be connected to the user's process using the dial facility. For more information on the dial facility, see the dial command in the Commands Manual or the dial_manager_ subroutine in the Subroutines Manual.

DATA TRANSFORMATION

One of the most visible functions of MCS is the transformation of data read from or written to the terminal. This may include rearrangement of white space, replacement of one character by a sequence of characters, and, in some cases, wholesale translation from one character code to another. The types of conversion for input and output are described in Section 3. The specific details of any particular conversion are determined by terminal type and, to a lesser extent, by the modes associated with the attachment. Terminal types are explained below, and the effects of the various modes are given in the description of the tty_ I/O module (see the Subroutines manual). The set_tty command, described in the Commands manual, can be used to change the terminal type or to modify many of the parameters used in converting input or output.

The special-purpose I/O modules (those other than tty_) usually perform their own data conversions independent of terminal type. They generally put the terminal in rawi and rawo modes (i.e., "raw" input and output) to prevent the rest of MCS from performing any transformations on data to or from the terminal.

TERMINAL TYPE CONCEPT

A terminal type is a named set of parameters identifying the characteristics and behavior of a terminal. The following attributes are components of a terminal type:

- character set (e.g., EBCDIC, ASCII, etc.)
- in response to carriage movement characters
- in response to other control sequences
- required for carriage movement functions (delays)
- control of horizontal tabs
- line length and page length

These parameters are used by the Multics Communication System to determine how to format output to, and interpret input from, the terminal. The specification of these individual parameters can be changed independently; the terminal type provides a mechanism for specifying them all at once without having to know the details of their implementation.

Terminal Type and Line Type

It is important to distinguish between terminal type and line type, both of which terms are used in describing a terminal connection to Multics. A line type defines the communications protocol used on a particular channel; it is a characteristic of a channel rather than of a terminal. The terminal type may be changed by the user in order to modify the system's treatment of the terminal; the line type is determined by the system, and cannot be changed while the channel is in use.

TERMINAL TYPE TABLE AND TERMINAL TYPE FILE

Terminal types are defined in a data base called the terminal type table (TTT). There is a system-wide TTT that is used by default; each process, however, can use its own TTT instead. The TTT being used by a process can be changed by means of the set_ttt_path command. The various entries of the ttt_info_ subroutine, described in the Subroutines manual, can be used to extract information from the TTT. The print_terminal_types command lists the names of all terminal types defined in the TTT; the display_ttt command displays the contents of the TTT in readable format. These commands are all described in the Commands manual.

The TTT is derived from an ASCII segment, suitable for creation and modification using a text editor, called the terminal type file (TTF). A TTT is generated from a TTF by means of the cv_ttf command, also described in the Commands manual. The syntax of a TTF is described later in this appendix.

Setting Terminal Types

Every terminal connected to the Multics system has a terminal type associated with it at all times. The terminal type associated with a particular terminal may be set in any of the following ways:

1. When the terminal dials up (i.e., a connection is established), its terminal type is set in accordance with its line type and baud rate as specified in the default type table in the TTT (see "Syntax of the TTF" below).

2. If the channel on which the terminal dialed up has an initial terminal type associated with it in the channel definition table (CDT), that terminal type is assigned to the terminal. See the MAM Communications for more information on the CDT.

3. If the terminal provides an answerback sequence that matches one of the answerback specifications in the TTT (see "Syntax of the TTF"), its terminal type is set according to the answerback.

4. If the user specifies the -terminal_type control argument to the login command or uses the terminal_type preaccess request, the terminal type is set accordingly. See the description of the login and terminal_type pre-access requests in the Commands manual.

5. The user may, at any time, change his terminal type by invoking the set_tty command with the -terminal_type control argument.

Changing Terminal Type Definitions

A user wishing to invent a new terminal type, or change the characteristics of an existing terminal type, may edit a copy of the system-supplied TTF and create a new TTT by using the cv_ttf command. Whenever he wishes to use the new or redefined terminal type(s), he switches to the new TTT by means of the set_ttt_path command, and then uses the set_tty command to change his own terminal type to the desired one. This change affects only his current process; other users of the same non-standard TTT are not affected until they use the set_tty command to set or change terminal type.

Note: Various sequences of characters beginning with the ASCII "escape" character (octal 033) are treated by some terminals, when sent as output, as commands to the terminal. These commands may have unexpected or undesirable effects on the behavior of the terminal if, for example, they are embedded in a piece of online mail. For this reason, the standard TTT distributed by Honeywell is designed to prevent the escape character from being included in normal output for most terminal types. Users or sites providing their own TTTs should be aware of the hazards of allowing escape sequences to be sent to terminals as a matter of course.

Terminal Type Table

The terminal type table (TTT), a data base that resides by default in the segment:

```
>system_control_1>ttt
```

describes all the terminal types used by MCS.

The TTT is a binary table containing numbers and pointers as well as character strings; therefore, it cannot be examined or modified using the editors. The display_ttt command is used to print out all or part of the TTT; when the system administrator wishes to add or delete terminal types, or change the information about one or more terminal types, he compiles a TTF into a TTT using the cv_ttf command, and then uses the install command to replace the copy of the TTT in the system.

A TTT is supplied by Honeywell that includes, but is not limited to, the following terminal types:

Terminal Type	Description
ASCIICAPS	Typical ASCII teleprinter terminal (uppercase only)
ASCII_CRT_CAPS	Typical ASCII crt terminal (uppercase only)
ADM3A	Lear Siegler Model ADM-3A
AJ630	Anderson-Jacobson Model 630
AMBASSADOR	Ann Arbor Ambassador CRT
CONCEPT100	Human Designed Systems Concept 100
DIABLO1640	Diablo Systems Series 1640
HAZELTINE1510	Hazeltine Model 1510
HEATH19	Heath Model H19
IBM3271	Control unit for IBM3270 terminal cluster
INFOTON100	Infoton 100 Display Terminal
IRISCOPE200	Iriscope 200
L6FTF	Honeywell L6 File Tranmission Facility
LA120	Digital Equipment LA120 DECwriter III
LED120	Triformation Systems braille terminal
NEC5520	Nippon Electric Model 5520 (Spinwriter)
NEC5525	Nippon Electric Model 5525 (Spinwriter)
SARA	Honeywell SARA 20
SYSTEM75	Selecterm System 75
TEK4023	Tektronix 4023
TEK4025	Tektronix 4025
TELERAY1061	Teleray 1061
TRANSLEX	ECD Translex Intelligent Terminal
TVI920	TeleVideo Model TVI-912 and 920
VIP7700_CLUSTER	Honeywell Multiple Interface Unit for Series VIP7700 Polled VIP Terminal
VIP7705	Honeywell VIP7700 Polled VIP Display Terminal (upper and lower case)
VIP7714	Honeywell VIP7714 read only printer
VIP7760	Honeywell VIP7760 Display Station
VIP7705R	Honeywell VIP7700R Polled VIP Display Terminal (upper and lower case)
VIP7760_CONTROLLER	Honeywell VIP7760 Controller
VIP7804	Honeywell VIP7804 Polled VIP Display Terminal
VIP7804_CLUSTER	Honeywell Multiple Interface Unit for Series VIP7804 Polled VIP Terminals
VISTAR	Infoton Vistar Satellite terminal
VT100	Digital Equipment Model VT-100

These terminal types can change at any time, so the user should invoke the print_terminal_types command to verify the current types.

SYNTAX OF THE TTF

The TTF defines all terminal types known to the system. It is an ASCII file which, when compiled into a binary table (the TTT), is installed at the system administrator's request.

The TTF consists of a series of entries describing terminal types, tables, and answerback interpretations. Each entry consists of a series of statements that begin with a keyword and end with a semicolon. White space and comments written in the same style as PL/I comments enclosed by /* and */ may appear between any tokens in the TTF. The last entry in the TTF must be the end statement. Global statements specifying defaults may appear anywhere before the end statement; the defaults they specify are in effect for all subsequent terminal type entries, until they are overridden by subsequent global statements. Except for the end statement, all statements consist of the statement keyword, a colon, the variable field of the statement, and a semicolon.

Generalized Character Specifications

Many statements in the TTF take as arguments single characters, or lists of single characters. Statements that accept such operands are shown with the <tty_char>notation. A <tty_char> operand may be any of the following:

1. A single unquoted character, such as X, A, p, $ or ~. This notation is only allowed for "simple" characters. This notation may *not* be used for control characters, white space, ASCII digit characters, "(", ")", "<", ">", " ", ":", ",", ";", or the double quote character.

2. A single quoted character, such as "X", ";", "B", or "0". Any ASCII code can be entered this way. Note that digits should be specified as "0", not 0.

3. A 1 to 3 digit octal number, such as 177, 14 or 007. This enters the character whose octal representation is as specified. Note that 0 is interpreted as octal 000. If the ASCII digit "0" is desired, it must be specified as "0" or 060.

4. The name of a control character, such as DEL. These may be either upper or lower case. All standard control characters are accepted, including:

```
NUL SOH STX ETX EOT ENQ ACK BEL (000 - 007)
BS  TAB LF  VT  FF  CR  SO  SI  (010 - 017)
DLE DC1 DC2 DC3 DC4 NAK SYN ETB (020 - 027)
CAN EM  SUB ESC FS  GS  RS  US  (030 - 037)
```

In addition, SP (040), DEL (177), NL (012), and HT (011) are also accepted.

5. Control characters may also be entered in the form ^A, which is read as control-A, and is the character sent when the control-A function is used on an ASCII keyboard. ^A is equivalent to SOH, or 001. The letters A-Z (upper or lower case equivalent) preceded by a "^" may be used for 001 through 032. Also accepted are ^@ (000), ^[(033), ^\ (034), ^] (035), ^^ (036), and ^_ (037).

Terminal Type Entry

The entry for each terminal type consists of a terminal_type statement naming the terminal type, followed by various statements describing the attributes of that terminal type. Attributes not specified for a terminal type are set from the defaults established by global statements or supplied by the cv_ttf command.

A description of each statement found in a terminal type entry is given below.

`terminal_type: <type name> {like <type name>};`

The terminal_type statement is required. It specifies the name of the terminal type described by the statements following it. The type name has a maximum length of 32 characters. All lowercase letters in the type name are translated to uppercase before being stored in the TTT. If the optional like keyword is supplied, it indicates that the attributes of the current terminal type are to be copied from the entry for the type whose name follows the like keyword, except for those that are overridden by subsequent statements in the current entry. The like keyword must refer to a previously defined terminal type.

`modes: <mode1>, <mode2>, ... <modeN>;`

The modes statement is required. It specifies the modes to be set when the type of the terminal is assigned. A mode name may be preceded by a ^ character to indicate that the specified mode is off for the terminal type. The line-length specification (li) must be included in the modes statement. See "Modes Operation" below for a list of the valid modes.

`function_keys: <table name>;`

The function_keys statement is optional. It specifies the name of a function_key table (defined by a function_key_table entry) to be used for this terminal. If it is omitted, or the table name is a null string, the terminal is assumed to have no function keys.

`initial_string: <string>;`

The initial_string statement is optional. If present, it specifies a character string to be sent to the terminal in rawo mode in order to initialize certain physical characteristics of the terminal (e.g., to set its horizontal tabs). This string is sent either at dialup time, in response to a "send_initial_string" order, or when set_tty is invoked with the -initial_string control argument. The string is specified as one or more substrings. Each substring may be one of the following:

1. A quoted string; e.g., "sR". If a quoted string is to contain a quote character, that quote must be doubled. (e.g., "s""R" is s"R).

2. <tty-char>

3. (<decimal-integer>) <<substring> ... <substring>>

   where <decimal-integer> is a repetition factor enclosed in parentheses and followed by one or more substrings enclosed in angle brackets (<and>). For example:

   (10) <040 ETX>

   represents 10 repetitions of the two character sequence consisting of a space and an ETX character (octal 003).

additional_info: <string>;

The additional_info statement is optional. If provided, it specifies additional information which may be needed to run the terminal. This information is not interpreted by the standard terminal software, and is not passed to the supervisor; it may be used by a special I/O module used to run terminals of the current type. The format and contents of the string depend on the particular application; it may even be the pathname of a segment containing additional information. The string is specified in the same way as for the initial_string statement (above).

bauds: <baud1> <baud2> ... <baudN>;

can also be written as:

bps: <baud1> <baud2> ... <baudN>;

The bauds statement is required if any delay statements (see below) are provided, and it must precede all delay statements. It specifies the baud rates to which the values supplied in the delay statements apply. A specification of "other" in the bauds statement means that the corresponding values in the delay statements apply to all baud rates not specified. If "other" is not specified, then delay values of 0 are assumed for all baud rates not specified in the bauds statement. The following is a list of the baud rates that may be specified:

110	600	2400	9600
150	1200	4800	19200
300	1800	7200	

`cps: <cps1> <cps2> ... <cpsN>;`

The cps statement may be used in place of the bauds statement (above) to express terminal speeds in characters per second. The value stored in the TTT is the corresponding baud rate. The cps values that may be specified, and their corresponding baud rates, are listed below:

cps value	*baud rate*
10	110
15	150
30	300
60	600
120	1200
180	1800
240	2400
480	4800
720	7200
960	9600
1920	19200

`<delay keyword>: <value1> <value2> ... <valueN>;`

In each delay statement, the same number of values must be supplied as baud rates in the bauds, bps, or cps statement. Each value specifies the number of delays to be used for the character described by the delay keyword at the baud rate specified in the corresponding position in the bauds statement (see example below). The possible delay keywords are:

vert_nl_delays
: the number of delays to be sent with a newline operation ($-127 \leq$ `vert_nl_delays` ≤ 127). If it is negative, its absolute value is the minimum number of characters that must be transmitted between two linefeeds.

horz_nl_delays
: the variable number of delays to be sent for each column position traversed by a carriage return or a newline operation. This is a floating point number ($0 \leq$ `horz_nl_delays` ≤ 1).

const_tab_delays
: the minimum number of delays to be sent with a horizontal tab ($0 \leq$ `const_tab_delays` ≤ 127).

var_tab_delays
: the number of additional delays to be sent for each column position traversed by a horizontal tab. This is a floating point number ($0 \leq$ `var_tab_delays` ≤ 1).

backspace_delays
: the number of delays to be sent with a backspace ($-127 \leq$ `backspace_delays` ≤ 127). If it is negative, its absolute value is the number of delays to be output with the first backspace of a series only (or a single backspace).

vt_ff_delays
: the number of delays to be sent with a vertical tab or formfeed (0 ≤ vt_ff_delays ≤ 511).

Values of zero are assumed at all baud rates for any delay type not specified.

Example:

```
bauds:              110   150   300   1200   other;

vert_nl_delays:       2     3     6     24      30;

horz_nl_delays:      .1    .12   .2    .8       1;

const_tab_delays:     0     1     2     7       10;

var_tab_delays:      .1    .12   .2    .8       1;

backspace_delays:     0     0     1     3       6;

vt_ff_delays:         0     0     0     0       0;
```

The first column gives the complete set of delay values to be used at 110 baud; the second column gives the values to be used at 150 baud, etc.

`line_types: <line_type name1>, <line_type name2>, ... <line_type nameN>;`
: The line_types statement is optional. It specifies the names of the line types on which a terminal of the current type can be run. If it is omitted, the current terminal type can run on any line type.

`erase: <tty_char>;`
: The erase statement is optional. It specifies the erase character for the terminal type. If it is omitted, the # character is used.

`kill: <tty_char>;`
: The kill statement is optional. It specifies the kill character for the terminal type. If it is omitted, the @ character is used.

`line_delimiter: <character>;`
: Specifies the terminal's normal line delimiter character. The character must be specified as one to three octal digits in the terminal's input code (untranslated). This character defaults to 012 unless the line type is 2741 or 1050, in which case it defaults to 055.

`keyboard_addressing: yes/no;`
: The keyboard_addressing statement is optional. It indicates whether or not to do keyboard locking and unlocking for a terminal on a communications channel whose line type is ASCII. If it is not provided, a value of no is assumed. This attribute is ignored for channels of any other line type.

`print_preaccess_message: yes/no;`

The print_preaccess_message statement is optional. It indicates whether or not the answering service should print a message advising the user to enter a preaccess request if the user entered an unrecognized login word. It is useful in cases where the character code of the terminal may be different from what was expected. At present, only one possible preaccess message is defined, suitable for use with EBCD and Correspondence-code IBM 2741 terminals. If the print_preaccess_message statement is omitted, a value of no is assumed.

`conditional_printer_off: yes/no;`

The conditional_printer_off statement is optional. It indicates whether or not the answerback identification of the terminal should be used to determine whether the terminal is equipped with the printer-off feature. If yes is specified, a terminal of this type is assumed not to have printer-off unless it has an answerback ID beginning with a digit (0 to 9); otherwise, the existence of the printer-off feature is deduced from the presence or absence of a printer-off sequence in the special characters table (see below). This attribute is primarily useful for IBM 2741 terminals. If the conditional_printer_off statement is omitted, a value of no is assumed.

`input_conversion: <table name>;`

The input_conversion statement is optional. It specifies the name of a conversion table (defined by a conversion table entry) to be used in converting input from the terminal. If it is omitted, or the table name is a null string, no input conversion table is used.

`output_conversion: <table name>;`

The output_conversion statement is optional. It specifies the name of a conversion table (defined by a conversion table entry) to be used in converting output sent to the terminal. If it is omitted, or the table name is a null string, no output conversion table is used. If the terminal is connected to the system through the tty_ I/O module, the output_conversion table cannot contain more than 128 characters. If the terminal is connected to the window_io_ I/O module, the output_conversion table can contain 256 characters.

`special: <table name>;`

The special statement is optional. It specifies the name of a table (defined by a special table entry) to be used as a special characters table when converting input and output (see "Special Characters Table Entry" below). If it is omitted, or the table name is a null string, no special characters table is used. If an output conversion table whose entries are not all 0 is specified, a special characters table must also be specified in order for the terminal to function correctly.

`input_translation: <table name>;`

The input_translation statement is optional. It specifies the name of a table (defined by a translation table entry) used to translate input from the code of the terminal to ASCII. If it is omitted, or the table name is a null string, input is not translated.

`output_translation: <table name>;`

The output_translation statement is optional. It specifies the name of a table (defined by a translation table entry) used to translate output from ASCII to the code of the terminal. If it is omitted, or the table name is a null string, output is not translated.

`old_type: <number>;`

The old_type statement is optional. It may be used for compatibility purposes to specify the numeric value of the terminal type formerly predefined by the Multics Communication System that most closely corresponds to the terminal type described by this terminal type entry.

`framing_chars: <frame_begin> <frame_end>;`

The framing_chars statement is optional. If present, it specifies the framing characters generated by the terminal when sending frame input at channel speed. The `<frame_begin>` and `<frame_end>` are `<tty_chars>`'s as defined above. In the terminal's character code they represent the frame_begin and frame_end characters respectively (i.e., without translation). `<frame_begin>` can be NUL or 000 to indicate that there is no frame_begin character; in this case, all input in blk_xfer mode is treated as part of a frame.

The following statements define parameters for flow control to and from asynchronous terminals. For more information, see the discussion of flow control in Section 3.

`input_suspend: <tty_char>;`

The input_suspend statement is optional. If present, it specifies a character to be transmitted to the terminal in iflow mode in order to temporarily suspend input or, alternatively, a character that the terminal sends to inform the system that it is suspending input. In either case, input is restarted when the input_resume character (see below) is sent to the terminal. This feature is appropriate for use on certain terminals which do input at line speed. If the input_suspend statement is present, the input_resume statement must also be present.

`input_resume: <tty_char> {, timeout};`

The input_resume statement is optional, unless the input_suspend statement (above) is present. It specifies a character that, when sent to the terminal by the system while in iflow mode, causes it to resume temporarily suspended input. Depending on the terminal, the input_suspend character (above) may not be required. The timeout keyword, if supplied, indicates that the terminal may suspend input (as at the end of a tape record) without transmitting an input_suspend character, in which case it is the responsibility of the system to detect this situation and send the input_resume character after input has been suspended. If the input_resume statement is specified but the input_suspend statement is not, the input_resume statement must include the timeout keyword.

`output_suspend: <tty_char>;`

The output_suspend statement is optional. It may be used with terminals that implement a suspend_resume protocol for output flow control. If present, it specifies a character that the terminal transmits to cause the system to suspend output so that the terminal can empty its internal buffer. The character is only interpreted by the system in oflow mode. Output is restarted when the terminal sends the output_resume character (see below). If the output_suspend statement is specified, the output_resume statement must also be specified, and none of the output_end_of_block, output_acknowledge, and buffer_size statements may be specified.

`output_resume: <tty_char>;`

The output_resume statement is optional, unless the output_suspend statement is present. It specifies a character transmitted by the terminal to inform the system that output that was suspended in response to an output_suspend character (see above) can be resumed. If the output_resume statement is present, the output_suspend statement must also be specified, and none of the output_end_of_block, output_acknowledge, and buffer_size statements may be specified.

`buffer_size: <number>;`

The buffer_size statement is optional. It may be used with terminals that implement a block acknowledgement protocol for output flow control. If present, it specifies the size in characters of the terminal's output buffer, and is used to determine the maximum number of characters to be sent to the terminal at one time (in one transmission) in oflow mode. Each block of up to that number of characters is terminated by an output_end_of_block character (see below). The next block is not transmitted until the terminal sends an output_acknowledge character. If the buffer_size statement is specified, the output_end_of_block and output_acknowledge statements must also be specified, and neither the output_suspend nor the output_resume statement may be specified.

`output_end_of_block: <tty_char>;`

The output_end_of_block statement is optional. If it is present, it specifies a character to be appended to every output block, as described under the buffer_size statement above. If the output_end_of_block statement is specified, the output_acknowledge and buffer_size statements must also be specified, and neither the output_suspend nor the output_resume statement can be specified.

`output_acknowledge: <tty_char>;`

The output_acknowledge statement is optional. If present, it specifies a character that is transmitted by the terminal when it is ready to receive the next block of output, as described under the buffer_size statement (above). If the output_acknowledge statement is specified, the buffer_size and output_end_of_block statements must be specified, and neither the output_suspend nor the output_resume statement may be specified.

Video Table Definition

Each terminal type may have an optional video table defined. This table contains control sequences for performing standard operations on video terminals. The table starts with the keyword:

```
video_info:
```

A global video table, which will be used for all terminal types that do not have a video table specified, is started with the keyword:

```
Video_info:
```

The absence of a video table may be specified by:

```
video_info:  ;
```

This may be used to negate the effects of a global Video_info statement or a video table inherited from a similar terminal type.

The video_info keyword is followed by 1 or more video info statements, described below. The video table is terminated by the first statement not in this list.

```
screen_height: <decimal-integer>;
```

specifies the usable number of lines on the screen.

```
screen_line_length: <decimal-integer>;
```

specifies the usable number of columns on the screen.

The following statements describe various video control sequences. Each <video_sequence> is a character string built by the concatenation of all the operands given. The sequence may also be followed by an optional delay or padding specification. Video sequences may be built out of any combination of the following:

```
<tty_char>
quoted string, such as "sR"
<addressing|repeat specification>
```

The addressing or repeat specification is entered as follows:

```
binary              LINE            +
(decimal {n}        COLUMN          - <ttychar})
octal {n}             N
```

This specification takes the value to be sent to the terminal (LINE, COLUMN, N), encodes it in some way (binary, decimal, octal), and adds or subtracts a fixed offset (+|- <tty-char>).

LINE represents the vertical or row position on the screen (1 origin). COLUMN represents the horizontal or column position on the screen (also 1 origin). The upper left hand corner of the screen, usually called home, is location LINE=1, COLUMN=1. The LINE and COLUMN notations are usually used in the absolute cursor addressing sequence, although they may be wherever required, depending on the terminal. N refers to a repeat count, which some terminals support for some operations.

These values may be encoded in either binary, decimal, or octal. Binary means byte (X), as in the PL/1 builtin. Decimal or octal causes the value to be converted to a character string representation. If {n} is given, it must be 1, 2, or 3, and refers to the length of the character string to be sent, padded with leading zeroes if required. If {n} is 0, or not specified, no leading zeroes will be sent. For example, if COLUMN is 35,

```
(decimal 3 COLUMN)      ->      "035"
(decimal COLUMN)        ->      "35"
(octal 3 COLUMN)        ->      "043"
(binary COLUMN)         ->      "#"
(COLUMN)                ->      "#"
```

If an offset is required, it may be specified as +|- <tty-char>. The value rank (tty-char) will be added to or subtracted from the number to be sent before it is encoded. A common example is (LINE + 037). In this case, a LINE of 1 will yield a space (octal 40), a LINE of 2 will yield " " (octal 41), etc.

Any video sequence may have an optional <padding> value, expressed as follows:

```
, pad n {us|ms}
```

If us (micro seconds), or ms (milliseconds) is specified, n is interpreted as a time value. Otherwise, it is an absolute number of pad characters required, regardless of the baud rate. If a time is specified, the minimum that can be specified is 100 microseconds. All values are rounded up to the next multiple of 100 microseconds. The maximum value is 26.2 seconds. Time values are converted to a pad count at execution time, depending on the baud rate of the terminal.

The following statements all use the syntaxes just described. Each statement also has a definition of exactly what effect the sequence has on the terminal. If the terminal does not have the capability to perform the function described, the statement should be omitted.

```
abs_pos:  <video-sequence> {<padding>} ;
```

defines the absolute cursor positioning sequence. This sequence moves the cursor to a given (LINE, COLUMN). Other than the cursor, no characters on the screen are affected.

```
clear_screen:  <video-sequence> {<padding>} ;
```

defines the screen clearing sequence. This sequence clears the entire screen to spaces regardless of where the cursor is, and leaves the cursor at home. This sequence does not clear tabs.

```
clear_to_eos:  <video-sequence> {<padding>} ;
```

defines the clear to end of screen sequence. This clears the screen from the current cursor position to the end of the screen. It does not move the cursor or clear tabs.

```
home:  <video-sequence> {<padding>} ;
```

defines the move cursor home sequence. The cursor moves to location LINE=1, COLUMN=1.

```
clear_to_eol:  <video-sequence> {<padding>} ;
```

defines the clear to end of line sequence. Starting at the current cursor position, the rest of the current line clears to spaces. The cursor does not move.

```
cursor_up:  <video-sequence> {<padding>} ;
```

defines a sequence to move the cursor up one row. It does not have any effect on the column. The effect of the sequence when the cursor is on the top line of the screen is undefined.

```
cursor_right:  <video-sequence> {<padding>} ;
```

defines a sequence to move the cursor one column to the right. It does not have any effect on the row. The effect of the sequence when the cursor is in the last column of the screen is undefined.

```
cursor_down:  <video-sequence> {<padding>} ;
```

defines a sequence to move the cursor down one row. It does not have any effect on the column. The effect of the sequence when the cursor is on the bottom line of the screen is undefined.

```
cursor_left:  <video-sequence> {<padding>} ;
```

defines a sequence to move the cursor one column to the left. It does not have any effect on the row. The effect of the sequence when the cursor is in the leftmost column of the screen is undefined.

```
insert_chars:  <video-sequence> {<padding>} ;
```

defines a sequence for inserting characters on the current line. If end_insert_chars (see next statement) is defined, insert_chars should put the terminal in a mode in which each character sent to the terminal is placed on the screen at the cursor location; each character to the right of the cursor is pushed one position to the right; and the cursor is moved one position to the right. The effect of pushing characters off the righthand edge of the screen is undefined. If end_insert_chars is not defined, insert_chars is defined as opening up N (or 1) spaces on the line, pushing characters to the right of the cursor toward the right. The cursor does not move in this case.

```
end_insert_chars:  <video-sequence> {<padding>} ;
```

defines a sequence for taking the terminal out of insert_chars mode. See above.

```
delete_chars:  <video_sequence> {<padding>} ;
```

defines a sequence for deleting characters from the current line. The character at the cursor is deleted, and all characters to the right are moved one column to the left. A space is inserted in the last column of the screen.

```
insert_lines:  <video_sequence> {<padding>} ;
```

defines a sequence for inserting lines on the screen at the current cursor position. All lines starting at the current line are moved down one line. The current line is filled with spaces. The effect of pushing lines off the bottom of the screen is not defined. This sequence is only defined to work when the cursor is at the leftmost margin. The position of the cursor is not changed.

```
delete_lines:  <video_sequence> {<padding>} ;
```

defines a sequence for deleting lines from the screen. The current line is deleted by moving all lines below it up one line. The bottom line of the screen is filled with spaces. This sequence is only defined to work when the cursor is at the leftmost margin. The position of the cursor is not changed.

Many terminals do not support all the functions described above, but often they can be simulated by combinations of other functions. For example, the Honeywell VIP7801 does not support clear_screen, as defined, because the clear sequence to that terminal also clears the tabs. The effect of this can be simulated, however, by the combination home (or abs_pos to 1,1) and clear_to_eos, which will clear the screen without affecting the tabs. Thus a clear_screen sequence could be defined which is a concatenation of the other two sequences. Similarly, if a terminal did not have a cursor up sequence, but did support abs_pos, it would be possible to specify a cursor_up sequence as a variant of the abs_pos sequence (by changing the offset by 1). In general, it is *not* recommended that this sort of optimization be done in the TTF. Instead, the TTF should be viewed as describing the physical characteristics of the terminal, and it is the job of software to choose from among the capabilities of the terminal in order to provide the desired effect.

For most applications, a certain minimal set of functions is required to perform video functions. These are:

1. Some way of clearing the screen. Clear_screen is best, but home and clear_to_eos will work, as well as erase_to_eol on each line.

2. Some way of absolute cursor addressing. Abs_pos is best, but the combination of home and the four cursor motion functions (up, down, left, and right) will work also.

The video_info entry for the Honeywell VIP 7801 is:

video_info:

```
screen_line_length:  80;
screen_height:       24;
home:                ESC H;
clear_to_eos:        ESC J, pad 1;
cursor_up:           ESC A;
cursor_right:        ESC C;
cursor_down:         LF;
cursor_left:         BS;
clear_to_eol:        ESC K;
insert_chars:        ESC "[1";
end_insert_chars:    ESC "[J";
delete_chars:        ESC "[P";
insert_lines:        ESC "[L";
delete_lines:        ESC "[M";
abs_pos:             ESC f (LINE + 037) (COLUMN + 037)
```

Modes Operation

The modes operation is supported when the I/O switch is open. The recognized modes are listed below. Some modes have a complement indicated by the circumflex character (^) that turns the mode off (e.g., ^erkl). For these modes the complement is displayed with the mode. Normal defaults are indicated for those modes that are generally independent of terminal type. The modes string is processed from left to right. Thus, if two or more contradictory modes appear within the same modes string, the rightmost mode prevails.

`8bit, ^8bit`
causes input characters to be received without removing the 8th (high-order) bit, which is normally interpreted as a parity bit. This mode is valid for HSLA channels only. (Default is off.)

`blk_xfer, ^blk_xfer`
specifies that the user's terminal is capable of transmitting a block or "frame" of input all at once in response to a single keystroke. The system may not handle such input correctly unless blk_xfer mode is on and the set_framing_chars order has seen issued. (Default is off.)

`breakall, ^breakall`
enables a mode in which all characters are assumed to be break characters, making each character available to the user process as soon as it is typed. This mode only affects get_chars operations. (Default is off.)

`can, ^can`
performs standard canonicalization on input. (Default is on.)

can_type=overstrike, can_type=replace
specifies the method to be used to convert an input string to canonical form. Canonicalization is only performed when the I/O switch is in "can" mode. (Default is can_type=overstrike.)

capo, ^capo
outputs all lowercase letters in uppercase. If edited mode is on, uppercase letters are printed normally; if edited mode is off and capo mode is on, uppercase letters are preceded by an escape (\) character. (Default is off.)

crecho, ^crecho
echoes a carriage return when a line feed is typed. This mode can only be used with terminals and line types capable of receiving and transmitting simultaneously.

ctl_char, ^ctl_char
specifies that ASCII control characters that do not cause carriage or paper motion are to be accepted as input, except for the NUL character. If the mode is off, all such characters are discarded. (Default is off.)

default
is a shorthand way of specifying erkl, can, ^rawi, ^rawo, ^wakes tbl, and esc. The settings for other modes are not affected.

echoplex, ^echoplex
echoes all characters typed on the terminal. The same restriction applies as for crecho; it must also be possible to disable the terminal's local copy function.

edited, ^edited
suppresses printing of characters for which there is no defined Multics equivalent on the device referenced. If edited mode is off, the 9-bit octal representation of the character is printed. (Default is off.)

erkl, ^erkl
performs "erase" and "kill" processing on input. (Default is on.)

esc, ^esc
enables escape processing (see "Typing Conventions" in Section 2) on all input read from the device. (Default is on.)

force
specifies that if the modes string contains unrecognized or invalid modes, they are to be ignored and any valid modes are to be set. If force is not specified, invalid modes cause an error code to be returned, and no modes are set.

`fulldpx, ^fulldpx`

allows the terminal to receive and transmit simultaneously. This mode should be explicitly enabled before enabling echoplex mode.

`hndlquit, ^hndlquit`

echoes a newline character and performs a resetread of the associated stream when a quit signal is detected. (Default is on.)

`iflow, ^iflow`

specifies that input flow control characters are to be recognized and/or sent to the terminal. The characters must be set before iflow mode can be turned on.

`init`

sets all switch type modes off, sets line length to 50, and sets page length to zero.

`lfecho, ^lfecho`

echoes and inserts a line feed in the user's input stream when a carriage return is typed. The same restriction applies as for crecho.

`lln, ^ll`

specifies the length in character positions of a terminal line. If an attempt is made to output a line longer than this length, the excess characters are placed on the next line. If ^ll is specified, line length checking is disabled. In this case, if a line of more than 255 column positions is output by a single call to iox_$put_chars, some extra white space may appear on the terminal.

`no_outp, ^no_outp`

causes output characters to be sent to the terminal without the addition of parity bits. If this mode and rawo mode are on, any 8-bit pattern can be sent to the terminal. This mode is valid for HSLA channels only. (Default is off.)

`oddp, ^oddp`

causes any parity generation that is done to the channel to assume odd parity. Otherwise, even parity is assumed for line types other than 2741 and 1050. This mode is valid for HSLA channels only. (Default is off.)

`oflow, ^oflow`

specifies that output flow control characters are to be recognized when sent by the terminal. The characters and the protocol to be used must be set before oflow mode can be turned on.

pl*n*, ^pl
: specifies the length in lines of a page. When an attempt is made to exceed this length, a warning message is printed. When the user types a formfeed or newline character (any break character), the output continues with the next page. The warning message is normally the string "EOP", but can be changed by means of the set_special control order. The string is displayed on a new line after *n* consecutive output lines are sent to the screen (including long lines which are folded as more than one output line). To have the end-of-page string displayed on the screen without scrolling lines off the top, *n* should be set to one less than the page length capability of the screen, unless the end-of-page string is a null string. In this case, output stops at the end of the last line of the page or screen. If ^pl is specified, end-of-page checking is disabled. (See description of scroll mode below.)

polite, ^polite
: does not print output sent to the terminal while the user is typing input until the carriage is at the left margin, unless the user allows 30 seconds to pass without typing a newline. (Default is off.)

prefixnl, ^prefixnl
: controls what happens when terminal output interrupts a partially complete input line. In prefixnl mode, a newline character is inserted in order to start the output at the left margin; in ^prefixnl mode, the output starts in the current column position. (Default is on.) Polite mode controls when input may be interrupted by output; prefixnl controls what happens when such an interruption occurs.

rawi, ^rawi
: reads the data specified from the device directly without any conversion or processing. (Default is off.)

rawo, ^rawo
: writes data to the device directly without any conversion or processing. (Default is off.)

red, ^red
: sends red and black shifts to the terminal.

replay, ^replay
: prints any partial input line that is interrupted by output at the conclusion of the output, and leaves the carriage in the same position as when the interruption occurred. (Default is off.)

`scroll, ^scroll`
: specifies that end-of-page checking is performed in a manner suited to scrolling video terminals. If the mode is on, the end-of-page condition occurs only when a full page of output is displayed without intervening input lines. The mode is ignored whenever end-of-page checking is disabled. (Default is off.)

`tabecho, ^tabecho`
: echoes the appropriate number of spaces when a horizontal tab is typed. The same restriction applies as for crecho.

`tabs, ^tabs`
: inserts tabs in output in place of spaces when appropriate. If tabs mode is off, all tab characters are mapped into the appropriate number of spaces.

`vertsp, ^vertsp`
: performs the vertical tab and formfeed functions, and sends appropriate characters to the device. Otherwise, such characters are escaped. (Default is off.)

`wake_tbl, ^wake_tbl`
: causes input wakeups to occur only when specified wakeup characters are received. Wakeup characters are defined by the set_wakeup_table order. This mode cannot be set unless a wakeup table has been previously defined.

Global Statements

A global statement specifies a default value for a terminal type attribute. It has the same form as the statement describing the attribute in a terminal type entry, except that the statement keyword begins with a capital letter. Global statements may not appear within terminal type entries. Global statements may be used for any of the statements listed above for a terminal type entry, except for terminal_type, initial_string, additional_info, and the delay statements. (A global Bauds, Bps, or Cps statement is allowed, although a global delay statement is not.) A global video table definition may be given by using the statement:

Video_info:

followed by one or more video table entries. The statement:

Video_info: ;

may be used to specify that no default video table exists.

Conversion Table Entry

A conversion table entry consists of two statements: one specifying the name of the table and one specifying its contents. The following is a description of a conversion table entry.

```
conversion_table:  <table name>;
<value0> <value1> ...  <value255>;
```

The table name is a string of up to 32 characters. The values are octal numbers of one to three digits; each value is the indicator corresponding to the character whose ASCII value is the index of the indicator in the table. The set_input_conversion and set_output_conversion orders to the tty_ I/O module (see the Subroutines manual) are presented below as a description of conversion tables and the indicators they contain. If fewer than 256 values are supplied, the unspecified values are assumed to be zero.

set_input_conversion

provides a table to be used in converting input to identify escape sequences and certain special characters. The info_ptr points to a structure of the following form: (defined in tty_convert.incl.pl1)

```
dcl 1 cv_trans_struc   aligned,
      2 version          fixed bin,
      2 default          fixed bin,
      2 cv_trans         aligned,
        3 value          (0 : 255) fixed bin(8) unaligned;
```

where version, default, and value are as described in the cv_trans_struc structure used with the set_input_translation order above. The table is indexed by the ASCII value of each input character (after translation, if any), and the corresponding entry contains one of the following values: (Mnemonic names for these values are defined in tty_convert.incl.pl1)

```
0 --  ordinary character
1 --  break character
2 --  escape character
3 --  character to be thrown away
4 --  formfeed character (to be thrown away if page length
      is nonzero)
5 --  this character and immediately following character to
      be thrown away
```

set_output_conversion

provides a table to be used in formatting output to identify certain kinds of special characters. The info_ptr points to a structure like that described for set_input_conversion (above). The table is indexed by each ASCII output character (before translation, if any), and the corresponding entry contains one of the following values: (Mnemonic names for these values are defined in tty_convert.incl.pl1)

```
0 --  ordinary character
1 --  newline
2 --  carriage return
3 --  horizontal tab
4 --  backspace
5 --  vertical tab
6 --  formfeed
7 --  character requiring octal escape
8 --  red ribbon shift
9 --  black ribbon shift
10 -- character does not change the column position
11 -- this character together with the following one do not
      change the column position (used for hardware escape
      sequences)
12 -- character is not to be sent to the terminal
17 -- or greater a character requiring a special escape
      sequence.  The indicator value is the index into the
      escape table of the sequence to be used, plus 16.  The
      escape table is part of the special characters table;
      see the set_special order below.
```

Translation Table Entry

A translation table entry consists of a statement specifying the name of the table and a statement specifying its contents, as described below.

```
translation_table:  <table name>;
<value0> <value1> ...  <value255>;
```

The table name is a string of up to 32 characters. The values are octal numbers of one to three digits. Each value is the result of translation of the character whose bit representation is the index into the table of that value (i.e., <value0> is the result of translating a character represented as 000, <value8> corresponds to a character represented as 010, etc.). The set_input_translation and set_output_translation orders to the tty_ I/O module (see the Subroutines manual) are presented below as a description of the translation tables and the indicators they contain. If fewer than 256 values are supplied, the unspecified values are assumed to be zero.

set_input_translation

provides a table to be used for translation of terminal input to ASCII. The info_ptr points to a structure of the following form: (defined in tty_convert.incl.pl1)

```
dcl 1 cv_trans_struc   aligned,
      2 version          fixed bin,
      2 default          fixed bin,
      2 cv_trans         aligned,
        3 value          (0 : 255) char(1) unaligned;
```

where:

version

is the version number of the structure. It must be 1.

default

indicates, if nonzero, that the default table for the current terminal type is to be used, and the remainder of the structure is ignored.

values

are the elements of the table. This table is indexed by the value of a typed input character, and the corresponding entry contains the ASCII character resulting from the translation. If the info_ptr is null, no translation is to be done.

NOTE: In the case of a terminal that inputs 6-bit characters and case-shift characters, the first 64 characters of the table correspond to characters in lower shift, and the next 64 correspond to characters in upper shift.

set_output_translation

provides a table to be used for translating ASCII characters to the code to be sent to the terminal. The info_ptr points to a structure like that described for set_input_translation (above). The table is indexed by the value of each ASCII character, and the corresponding entry contains the character to be output. If the info_ptr is null, no translation is to be done.

NOTE: For a terminal that expects 6-bit characters and case-shift characters, the 400(8) bit must be turned on in each entry in the table for a character that requires upper shift and the 200(8) bit must be on in each entry for a character that requires lower shift.

Function Key Table Entry

A function key table is begun and named by a function_key_table statement, which is the only required statement. All the remaining statements define function key sequences, and are optional. A function key is defined by giving the name of the key, and the characters transmitted when the key is struck. The following names are recognized: home, up, down, left, right, and key(i), where i must be 0 or greater, and is the number of the function key. If the terminal has no function key labelled 0, then the first key may be 1. No gaps are permitted, but the keys may be defined in any order.

Up to four sequences may be defined for each key, giving the sequences transmitted for the function key, the function key when shifted, the function key when the control key is held down, and the function key with both shift and control, in that order, separated by commas, and terminated by a semi-colon. If less than four sequences are given, or a sequence is missing, the terminal is assumed to not have a function key for that combination of key-strokes.

If the terminal always takes some local action (e.g. clearing the screen, moving the cursor) (possibly in addition to transmitting the sequence) when a key is struck, it is better to omit the sequence entirely, since most applications will not want the side-effect to occur, and would most likely not even use the key.

Example

```
function_key_table: vip7801_function_keys;
home: ESC H;
left: ESC D;
right: ESC C;
up: ESC A;
down: ESC B;
key(0): ESC e, ESC `, ESC c;
key (1): ESC 0, ESC 1;
key (2): ESC 2, ESC 5;
key (3): ESC 6, ESC 7;
key (4): ESC 8, ESC 9;
key (5): ESC :, ESC ";";
key (6): ESC <, ESC =;
key (7): ESC >, ESC ?;
key (8): ESC P, ESC Q;
key (9): ESC R, ESC S;
key (10): ESC T, ESC V;
key (11): ESC \, ESC ];
key (12): ESC ^, ESC _;
```

Special Characters Table Entry

A special characters table entry consists of a special_table statement and a set of statements specifying the contents of a special characters table. These statements are described below. Wherever the expression <sequence> appears, it means from zero to three <tty_char>s, separated by white space, representing a sequence of characters to be output to fulfill the specified function. If any statement specifying a sequence is omitted, a null sequence is assumed, unless otherwise specified in the description of the statement. All sequences are in ASCII code except for the printer_on and printer_off sequences. For those sequences that are used when specific indicators are encountered in the output conversion table, the relevant indicator is given in the description of the statement. See the description in the Subroutines manual of the various tables in the discussion of orders to the tty_ I/O module for more detailed information.

`special_table: <table name>;`

The special_table statement specifies the name of the table. It is a string of up to 32 characters.

`new_line: <sequence>;`

The new_line statement specifies the sequence to be output for a newline character (output conversion indicator 1).

`carriage_return: <sequence>;`

The carriage_return statement specifies the sequence to be output for a carriage return character (output conversion indicator 2). If the sequence is null, backspaces are used to move the carriage to the left margin.

`backspace: <sequence>;`

The backspace statement specifies the sequence to be output for a backspace character (output conversion indicator 4). If the sequence is null, a carriage return and spaces are used to reach the correct column. The carriage return and backspace sequences should not both be null.

`tab: <sequence>;`

The tab statement specifies the sequence to be output for a horizontal tab character. If the sequence is null, an appropriate number of spaces is used to reach the next tab stop.

`vertical_tab: <sequence>;`

The vertical_tab statement specifies the sequence to be output for a vertical tab character (output conversion indicator 5) if the terminal is in vertsp mode.

`form_feed: <sequence>;`

The form_feed statement specifies the sequence to be output for a formfeed character (output conversion indicator 6) if the terminal is in vertsp mode.

`printer_on: <sequence>;`

The printer_on statement specifies the sequence to be output to fulfill a "printer_on" order. The sequence is specified in the character code of the terminal. If the sequence is null, the printer_on feature is not supported.

`printer_off: <sequence>;`

The printer_off statement specifies the sequence to be output to fulfill a "printer_off" order. The sequence is specified in the character code of the terminal. If the sequence is null, the printer_off feature is not supported.

`red_shift: <sequence>;`

The red_shift statement specifies the sequence to be output for a red-ribbon-shift character (output conversion indicator 10 (octal)).

`black_shift: <sequence>;`

The black_shift statement specifies the sequence to be output for a black-ribbon-shift character (output conversion indicator 11 (octal)).

```
end_of_page:  <sequence>;
```

The end_of_page statement specifies the sequence to be output when output is suspended because the page length of the terminal has been reached. If it is omitted, the character sequence "EOP" is assumed. A null string indicates that output is to stop at the right margin of the last line of a page.

```
output_escapes:  <indicator1> <sequence1>,
<indicator2> <sequence2>, ...  <indicatorN> <sequenceN>;
```

The output_escapes statement specifies the escape sequences to be output for characters whose output conversion indicators are 21 (octal) or greater when the terminal is in ^edited mode. The indicators specified in the statement are the same as the corresponding indicators in the output conversion table.

```
edited_output_escapes:  <indicator1> <sequence 1>,
<indicator2> <sequence2>, ...  <indicatorN> <sequenceN>;
```

The edited_output_escapes statement specifies sequences like those specified by the output_escapes statement, but they are used when the terminal is in edited mode.

```
input_escapes:  <value1> <result1>,
<value2> <result2>, ...  <valueN> <resultN>;
```

The input_escapes statement specifies those input characters that are to be interpreted as escape sequences when preceded by an escape character, and the resulting characters that replace those sequences. (An escape character in this context is a character defined by software to initiate an escape sequence, i.e., one with an indicator of 2 in the input conversion table.) Each "value" is an octal number representing the ASCII value of a character that is used in an escape sequence; the corresponding "result" is an octal number representing the single character that replaces the escape sequence in the input stream.

Default Types

Exactly one default_types statement must appear in the TTF. It specifies default terminal types on the basis of baud rate and line type. When a terminal dials up, this information is used by the answering service to assign its type if no default terminal type is specified in the CDT entry for the channel. The default_types statement is described below.

```
default_types:  <baud1> <line_type1> <terminal_type1>,
<baud2> <line_type2> <terminal_type2>, ...
<baudN> <line_typeN> <terminal_typeN>;
```

Each baud*i* is a number representing a baud rate, or the word "any"; each line_type*i* is the name of a valid line type, or the word "any"; each terminal_type*i* is the default terminal type for the specified combination of baud rate and line type. The table thus constructed is searched in the order in which the baud rate, line_type, terminal_type triplets are specified, and the first entry that matches the particular channel is used to determine the initial terminal type. The last entry in the table should specify "any" for both baud rate and line type.

Answerback Table

The answerback table consists of entries specifying how to determine a terminal type and identification on the basis of its answerback. The answerback sent by the terminal is scanned under control of each answerback table entry, starting with the first one specified in the answerback table. If the scan succeeds (as described below), and the line type of the terminal is one that is valid for the terminal type specified in the answerback table entry, the terminal type and ID are derived from that entry; otherwise, the answerback is rescanned using the next entry, and so on. An answerback table entry consists of two statements: an answerback statement and a type statement.

`answerback: <keyword1> <value1>, <keyword2> <value2>, ... <keywordN> <valueN>;`

The answerback statement describes how the scan of the answerback is to be performed. The "scan pointer," indicating the current character position in the answerback of the scan, starts at the beginning of the answerback string and is adjusted according to the controls specified by the answerback statement. The possible keyword-value pairs are described below.

`match <expression>`

`<expression>` is either the word "digit," the word "letter," or a string enclosed in quotes. If it is digit or letter, the scan fails unless the character addressed by the scan pointer is a digit (0 to 9) or a letter (A to Z or a to z), respectively. If it is a quoted string, the scan fails unless the scan pointer points to the beginning of a matching string. If the match succeeds, the scan pointer is advanced over the matching string or character, and the scan is continued using the next keyword-value pair.

`search <expression>`

works like match, except that the scan succeeds if the matching character or string is found anywhere to the right of the scan pointer.

skip N

causes the scan pointer to be moved N characters to the right. The value N may be negative, in which case the pointer is actually moved to the left. The scan fails if there are fewer than N characters between the scan pointer and the end (or beginning if N is negative) of the answerback string.

id N

the N characters starting at the right of the scan pointer form the ID of the terminal. The value N must be in the range 1 <= N <= 4. If there are fewer than N characters to the right of the scan pointer, the scan fails.

id rest

as many characters (up to 4) as remain to the right of the scan pointer constitute the ID of the terminal (not including control and carriage-motion characters).

type: <type name>;

The type statement specifies the name of the terminal type to be assigned to a terminal whose answerback satisfies the specification in the answerback statement. The specified terminal type must be defined by a previous terminal type entry. If the type statement is omitted, the answerback is to be used to set the ID only, and the terminal type is not changed.

Preaccess Commands

The preaccess command entries are used to define the terminal types to be set in response to preaccess commands at dialup time. Each preaccess command entry consists of a preaccess_command statement and a type statement. See the Commands manual for more information about preaccess commands.

preaccess_command: <command>;

The preaccess_command statement specifies the name of a preaccess command. Preaccess commands include help (HELP), MAP, hello, 963, 063, 029, modes, echo, and terminal_type (ttp). If a preaccess command statement is not present for any one of these command statements, the command statement has no effect when entered from the terminal.

type: <type name>;

The type statement specifies the terminal type to be assigned when the corresponding command is entered. The specified type must be defined by a previous terminal type entry.

Examples

```
/* Sample terminal type entries */

 Input_conversion:  standard_input_conv;
*

 terminal_type:  TN300;
  modes:  default,hndlquit,tabs,llll8;
  initial_string:  ESC "2" CR ESC "1" (11) < (10)  (SP)  ESC "1";
  bauds:                 110   150   300  1200;
  vert_nl_delays:          0     2     6   -38;
  backspace_delays:       -2    -3    -6   -27;
  vt_ff_delays:           19    29    59   230;
  output_conversion:  ascii_output_conv;
  special:  tn300_special;
  line_types:  ASCII, 202ETX;
  old_type:  4;
```

```
/* sample default_types statement and answerback entries */

default_types:
                    110   ASCII  TTY33,
                    any   ASCII  ASCII,
                    any     VIP  ASCII,                              *
                    any     any   G115;
                                                                     *
answerback:         search " E", id 3;
 type:              TN300;

answerback:         search " E";
 type:              TN300;


/* sample conversion, translation, and special tables */

conversion_table:  standard_input_conv;
          03 00 00 00 00 00 00 00
          00 00 01 00 04 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 05 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 02 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 00
          00 00 00 00 00 00 00 03;
                                                                     *
end;
```

NAMES OF COMMUNICATIONS CHANNELS

The name of a communications channel is an encoding of the information describing the physical connection. Every such name is a string of 1 to 32 characters. The name is divided into components separated by "." characters; each component represents a level of multiplexing.

The first two components have a standard form and describe a physical (FNP) channel on an FNP. Multiplexed channels (i.e., subchannels of a concentrator whereby multiple terminals are supported on a single FNP channel) have additional components identifying the individual subchannels. The form of each component depends on the type of multiplexer involved.

The general form of the name of a physical channel is:

```
F.ANSS
```

where:

F
: is a top-level multiplexer name. If this is an FNP, the name must be a, b, c, d, e, f, g, or h. Other system or user defined top-level multiplexers may have different naming conventions.

A
: is l for a channel of a low-speed line adapter (LSLA) or h for a channel of a high-speed line adapter (HSLA).

N
: is the number of the LSLA or HSLA on the specified FNP. It is in the range 0 to 5 for LSLAs or 0 to 2 for HSLAs.

SS
: is a 2-digit decimal number identifying a subchannel of the specified LSLA or HSLA.

T & D Channel

A channel called F.c000, where F is an FNP identifier, is a special virtual channel used by COLTS (Communications Online Test and Diagnostics System). It does not correspond to an actual physical channel on the FNP.

Examples

```
a.1003      FNP a, LSLA 0, subchannel 03
a.h219      FNP a, HSLA 2, subchannel 19
c.l411      FNP c, LSLA 4, subchannel 11
```

```
printer_on:  15;
printer_off:  16;

red_shift:  033 141;
black_shift:  033 142;
end_of_page:  105 117 120;

output_escapes:
    21 134 074,              /* esc < ([) */
    22 134 076,              /* esc > (]) */
    23 134 047,              /* esc . (^) */
    24 134 050,              /* esc ( ({) */
    25 134 051,              /* esc ) (}) */
    26 134 164;              /* esc t (~) */

edited_output_escapes:
    21 050 010 075,          /*  (= ({) */
    22 051 010 075,          /*  )= (}) */
    23 047,                  /*  .  (~) */
    24 050 010 055,          /*  (- ({) */
    25 051 010 055,          /*  )- (}) */
    26 047 010 136;          /*  .~ (~) */

Input_escapes:
    074 133,                 /* esc < -> [ */
    076 135,                 /* esc > -> ] */
    047 140,                 /* esc . -> ~ */
    050 173,                 /* esc ( -> { */
    051 175,                 /* esc ) -> } */
    164 176,                 /* esc t -> ~ */
    124 176;                 /* esc T -> ~ */
```

end;

NAMES OF COMMUNICATIONS CHANNELS

The name of a communications channel is an encoding of the information describing the physical connection. Every such name is a string of 1 to 32 characters. The name is divided into components separated by "." characters; each component represents a level of multiplexing.

The first two components have a standard form and describe a physical (FNP) channel on an FNP. Multiplexed channels (i.e., subchannels of a concentrator whereby multiple terminals are supported on a single FNP channel) have additional components identifying the individual subchannels. The form of each component depends on the type of multiplexer involved.

The general form of the name of a physical channel is:

```
F.ANSS
```

where:

F
: is a top-level multiplexer name. If this is an FNP, the name must be a, b, c, d, e, f, g, or h. Other system or user defined top-level multiplexers may have different naming conventions.

A
: is 1 for a channel of a low-speed line adapter (LSLA) or h for a channel of a high-speed line adapter (HSLA).

N
: is the number of the LSLA or HSLA on the specified FNP. It is in the range 0 to 5 for LSLAs or 0 to 2 for HSLAs.

SS
: is a 2-digit decimal number identifying a subchannel of the specified LSLA or HSLA.

T & D Channel

A channel called F.c000, where F is an FNP identifier, is a special virtual channel used by COLTS (Communications Online Test and Diagnostics System). It does not correspond to an actual physical channel on the FNP.

Examples

```
a.1003      FNP a, LSLA 0, subchannel 03
a.h219      FNP a, HSLA 2, subchannel 19
c.1411      FNP c, LSLA 4, subchannel 11
```

APPENDIX C

PUNCHED-CARD INPUT OUTPUT AND RETURNED OUTPUT CONTROL RECORDS

BULK DATA INPUT

Each deck must begin with three (or more) keypunched control cards described below. These cards are used to identify the submitter to the Multics system and specify the type of card input. The decks are then submitted to operations personnel, and, in general, are read in by the next day. For protection, segments are created in system pool storage rather than in the user's directory. Once the data has been read, the user may copy the card image segment into his directory with the copy_cards command (see the description of this command in the Commands manual.)

Card image segments must be copied from the system pool storage within a reasonable time, as these segments are periodically deleted.

Because different card readers have different punch card decoding conventions, the user is warned that the same character may require different punch codes on different readers. Thus, for example, a + character may be represented by a 12-8-2 punch on one reader and a 12-0 on another. Obviously, cards prepared for the first reader do not transmit the same data on the second reader and may in fact be unreadable. The user should consult the manufacturer's documentation before preparing any cards for input.

EXAMPLE

Suppose user Jones working on project Graybar, wishes to read a FORTRAN source deck into a segment called alpha.fortran, with an access class of "Proprietary, Accounting." The deck was created on a standard keypunch. The control cards might be as follows:

```
++DATA  ALPHA.FORTRAN  \JONES  \GRAYBAR
++PASSWORD  XXX
++FORMAT  RMCC  TRIM  LOWERCASE  ADDNL
++AIM  \PROPRIETARY,
++AIM  \ACCOUNTING
++INPUT
  .
  .
(Text of alpha.fortran)
  .
  .
```

The text of alpha.fortran above, is the user deck that contains the data.

The control cards followed by the user's data cards are usually submitted to operations personnel for reading. When the cards have been read into Multics by operations personnel, the user is notified by mail that his deck has been read in. He should then log in as Jones.Graybar (either interactively or by using the remote job entry facility) with an access class of "Proprietary, Accounting" and issue the command:

```
copy_cards  alpha.fortran
```

to copy the card image segment into the working directory. If the command does not succeed, then an error message is issued to explain the problem. The user may need to check with operations to correct the problem.

CONTROL CARDS FOR BULK DATA

In the following discussions of control cards, parameters to be entered by the user are shown as all uppercase characters. The user may keypunch either upper or lowercase characters when preparing card decks; internal conversion of uppercase to lowercase characters is forced by the system for all control cards. The escape convention is honored if the user wishes to input characters that are to remain uppercase. All letters punched on the control cards are mapped to lowercase except those immediately following an escape character (backslash or cent sign). For example, \SMITH.\SYS\MAINT is mapped into Smith.SysMaint.

The control card format is as follows:

1. it begins with ++ in columns 1 and 2
2. a keyword begins in column 3
3. balance of the card after the keyword is free form
4. continuation cards are not permitted; each control card must be contained within 80 columns
5. cards are read with lowercase, nocontin, noaddnl modes (see "Card Conversion Modes" later in this section).

Name: ++DATA

This control card tells the card input process that the deck is to be read as bulk data input. It must be the first control card of the deck. All three fields of this control card must be specified in the order shown.

Usage

```
++DATA <DECK_NAME> <PERSON_ID> <PROJECT_ID>
```

where:

1. DECK_NAME
   is the name used to separate each deck and to identify the card image segment in system pool storage. It should be unique among the user's decks recently submitted. In the event of name duplications, the system card reading process appends a numeric component to the end of the supplied name and creates a duplicate card image segment for DECK_NAME unhess the OVERWRITE control option is specified on the ++CONTROL card.

2. PERSON_ID
   is the registered person name of the submitter as used during login. Only this person is able to read the card image segment from the pool.

3. PROJECT_ID
   is the registered project name of the submitter.

NOTES

Multics person and project names normally begin with uppercase letters. Such names must have an escape character punched before each uppercase letter, since all letters punched on the control cards are mapped to lowercase except those immediately following an escape character (backslash or cent sign).

Angle brackets in the "Usage" line indicate information that is supplied by the user.

Name: ++PASSWORD

This control card is used to specify the user's card input password. It must be specified and must immediately follow the ++DATA card.

Usage

```
++PASSWORD <xxxxxxxx> {-control_arg}
```

where:

1. xxxxxxxx
   is the card input password registered for this user. This password is normally different from the user's login password. It is maintained by the system administrator. It is customary to turn off the printing mechanism on the keypunch when creating this card. Users who have r access to `>scl>rcp>card_input_password.acs` do not need to be registered for card input or type a card input password in order to have input accepted. In this case, xxxxxxxx should be blanks.

2. -control_arg
   may be -cpw STR to change the password, where STR is a new password of up to 8 characters.

Name: ++AIM

This control card is used to specify the AIM access class of the data on the cards in the user's deck. It is an optional control card and if not specified, an access class of system_low is assumed.

Usage

```
++AIM <ACCESS CLASS>
```

where ACCESS CLASS is the access class of the data. The access class field may contain embedded spaces and commas. If the complete access class does not fit onto a single card, additional ++AIM cards can be used. The access class fields of all the ++AIM cards must define a valid access class when concatenated in deck order. Trailing blanks are stripped off before concatenation is done. Concatenation to form a valid access class is performed on successive access class strings, separated by a blank.

The access authorization of the process that runs the remote device must be the same as the access class given in the ++AIM control card for the deck to be accepted by the system.

Name: ++FORMAT

This control card is used to define the punch code conversion used to interpret the data in the user's card deck (not control cards). It is an optional control card and, if omitted, the MCC punch code conversion is assumed for local card readers and RMCC mode for remote card readers.

Usage

```
++FORMAT <PUNCH_FORMAT> <MODES>
```

where:

1. PUNCH_FORMAT
   is the punch code conversion to use in reading the card deck. It must be either MCC, RMCC, VIIPUNCH, or RAW, all of which are described in Section 5. (Not all card readers support each of these conversion modes.) This field is required.

   NOTE: Most remote card readers are able to read in the RMCC conversion mode only.

2. MODES
   this field is optional and may be any of the following. It is meaningful only for MCC and RMCC formats. (Refer to the discussion of "Card Conversion Modes" for a description of these modes.)

```
TRIM            (default)
NOTRIM
LOWERCASE
NOCONVERT       (default)
ADDNL           (default)
NOADDNL
CONTIN
NOCONTIN        (default)
```

Name: ++CONTROL

This control card is used to control the way the card reading software operates. If the control string OVERWRITE is specified, then if the DECK_NAME specified on the ++DATA card already exists in the system card pool, the segment is truncated before input is started. This feature is useful should communication line error or operator error require multiple inputs of the same card deck. Also see ++CONTROL card for remote job entry. The feature is disabled if a blank password is used.

Usage

```
++CONTROL OVERWRITE
```

Name: ++INPUT

This control card marks the end of the control cards. The next card is the first card of the user's data to be placed in the card image segment. This card is required for all decks.

Usage

```
++INPUT
```

There are no fields following the key on this control card.

USER DATA CARDS

All user data cards following the ++INPUT card are copied into the card image segment. The data may consist of any card punch combinations acceptable for the specified punch code conversion mode, except for an end-of-file marker which is defined as a card with "++EOF" in columns 1 through 5, and blanks in columns 6 through 80. This end-of-file marker defines the end of the user's data. The ++EOF card is supplied by the operator. If the user supplies it, the card deck is not read in successfully and the card deck aborts.

REMOTE JOB ENTRY

Remote job entry (RJE) on Multics is a mechanism that allows a registered user to submit an absentee job via a card deck. The card deck must contain standard Multics commands exactly like an interactive user would put into an absentee input (absin) segment. The user's card deck is copied into an absentee input segment that is created in the normal system pool storage used for bulk data input. When the user's deck has been successfully read, an absentee request is submitted on behalf of the user specified in the deck.

A special header is added to the absentee input segment so that a dprint request of the absentee output segment is automatically given using the request type associated with the remote terminal used to read in the RJE card deck. The header consists of the following lines:

```
&command_line off
rje_args$set prt_rqt X
rje_args$set pun_rqt Y
rje_args$set station Z
set_epilogue_command "eor -dl -rqt [rje_args prt_rqt] [user absout]"
```

where:

X is the printer request type of the submitting station.

Y is the punch request type of the submitting station.

Z is the station ID of the submitting station.

If the remote terminal does not have a printer request type, the dprint of the absentee output segment is issued for the central site printer.

The absentee process is created as Person_id.Project_id.p at the AIM authorization specified on any ++AIM control cards. The absout file is put in the user's home directory, unless otherwise specified on the ++RJECONTROL card described below.

A complete card deck for Multics RJE (as the user would submit it to operations) is shown below.

The user identified on the ++RJE card is notified by mail when his absentee input card deck has been read and his RJE job has been successfully queued.

Example of Remote Job Entry

Suppose user Jones of project Graybar wishes to list the contents of all directories in his master_files subtree. These files have an access class of Proprietary, Accounting. He wishes the absentee job to be restarted in case of a crash and he wants it to be run in absentee queue 2. Since his process will be running at an authorization greater than system_low, he has specified that the output file be placed in his subdirectory pro_acc. The RJE input deck he would use might look like this:

```
++RJE  LIST_SUBTREE.ABSIN  \JONES  \GRAYBAR
++PASSWORD  ZZZ
++AIM  \PROPRIETARY, \ACCOUNTING
++RJECONTROL  -RT  -Q 2
++RJECONTROL -OF >UDD>\GRAYBAR>\JONES>PRO_ACC>LIST.ABSOUT
++RJEARGS  >UDD>\GRAYBAR>\JONES>MASTER_FILES
++INPUT
&PRINT  ^/\DIRECTORY \LISTING OF \SUBTREE:  &1^2/
WS &1 "LIST -A -DTCM -SORT"
&PRINT  DONE.
LOGOUT
```

As the last step in executing the logout command, the special commands placed in the header of his absin file cause the absentee output segment to be dprinted using the printer request type used for the submitting station. Note that the job has been constructed to list any subtree (to which the user has access) by simply replacing the ++RJEARGS control card using another pathname.

CONTROL CARDS FOR REMOTE JOB ENTRY

The following is a list of RJE control cards. The format is the same as for bulk data cards discussed in the "Control Cards for Bulk Data" section above.

Name: ++RJE

This control card tells the card input process that the deck is to be read as a set of RJE absentee commands and submitted as an absentee job for Person_id.Project_id. It must be the first card of the deck. All three fields of the control card must be specified in the order shown.

Usage

```
++RJE <DECK_NAME> <PERSON_ID> <PROJECT_ID>
```

where:

1. DECK_NAME
   is the name of the user's absentee input segment. If it does not end in ".absin", this suffix is supplied. The name should be unique among all RJE decks recently submitted by the user. In the event of name duplications, the card reading process adds a numeric component just preceding the .absin suffix and creates a duplicate absentee input segment for DECK_NAME.

2. PERSON_ID
   is the registered name of the submitter as used during login. This is the person name under which the absentee job is run.

3. PROJECT_ID
   is the registered project of the submitter. This is the project name under which the absentee job is run.

NOTES

Multics person and project names normally begin with upper case letters. Such names must have an escape character punched before each uppercase letter, since all letters punched on the control cards are mapped to lowercase except those immediately following an escape character (backslash or cent sign).

Name: ++PASSWORD

This control card is used to specify the user's card input password. It must be specified and must immediately follow the ++DATA card.

Usage

```
++PASSWORD <xxxxxxxx> {-control_arg}
```

where:

1. xxxxxxxx
   is the card input password registered for this user. This password is normally different from the user's login password. It is maintained by the system administrator. It is customary to turn off the printing mechanism on the keypunch when creating this card. A blank password is not allowed. For more information refer to the ++RJE or ++DATA control cards.

2. -control_arg
   may be -cpw STR to change the password, where STR is a new password of up to 8 characters.

Name: ++RJECONTROL

This control card is used to specify control arguments that can be given to the enter_abs_request command. It is an optional control card. Multiple ++RJECONTROL cards may be used if all the control arguments do not fit on a single card.

Usage

```
++RJECONTROL <ARG1> <ARG2> ...  <ARGn>
```

where ARGi is any control argument acceptable to the enter_abs_request command, except for -ag or -argument (see the ++RJEARGS control card). If multiple ++RJECONTROL cards are used, the order of the control arguments is the concatenation of each ARGi string (separated by a space) in deck order. A control argument must not be split across cards and must have leading hyphens where appropriate. Any pathnames specified in the control arguments must be absolute pathnames.

Name: ++RJEARGS

This control card is used to pass arguments to the absentee process, as would normally be done by using the -ag or -argument control argument to the enter_abs_request command. It is an optional control card. If there are more arguments than can fit on a single card, additional ++RJEARGS cards may be used. Arguments must not be split across cards.

Usage

```
++RJEARGS <ARG1> <ARG2> ...  <ARGn>
```

where ARGi is the ith argument to be passed to the absentee process (used in substitutions of the form &i). If multiple ++RJEARGS control cards are used, the order of the arguments is the concatenation of each ARGi string (separated by a space) in deck order.

Name: ++EPILOGUE

The ++EPILOGUE control card overrides the default command string:

```
dp -dl -rqt [rje_args prt_rqt] [user absout]
```

(which is executed at logout time) with the one supplied. This allows the user to control what action is taken just prior to logout of his absentee process.

Usage

```
++EPILOGUE <COMMAND_LINE>
```

where COMMAND_LINE is any command acceptable in an absentee process. If multiple ++EPILOGUE cards are used a single command line is generating by concatenating the values contained on the ++EPILOGUE cards separated by spaces.

Name: ++ABSIN

This control card allows the RJE submitter to use an already online absentee input segment instead of including one as an input deck. If any user-supplied cards follow the ++INPUT card the input is aborted.

Usage

```
++ABSIN <PATHNAME> {SYSTEM}
```

where PATHNAME is the absolute pathname of the absentee input segment. If the optional argument SYSTEM is specified, then pathname is assumed to be the entryname of the absentee input segment in `>system_library_tools`.

Name: ++FORMAT and ++INPUT

The ++FORMAT and ++INPUT cards are the same as for punched card input described above. The ++INPUT card is required at all times.

User Absentee Commands

All cards following the ++INPUT card for remote job entry are copied into the absentee input segment as commands. The command lines are translated according to the modes specified on the ++FORMAT card if present, or by the default modes which are TRIM, NOCONVERT, NOCONTIN, and ADDNL. (For more information see "Card Input Conversion Modes" below.) Any command lines may be given except for an end-of-file marker which is defined as a card with "++EOF" in columns 1 through 5, and spaces in columns 6 through 80. The end-of-file marker defines the end of the user's data and is supplied by the operator, not the user. If an end_of_file marker is supplied by the user, his card input will be aborted.

Card Formats

There are several ways to interpret the punched codes on cards. The user should generate his card deck according to the format that best meets his needs. The following formats are supported by Multics.

```
MCC
VIIPUNCH
RAW
RMCC
```

Punch codes are not specified, unlike MCC format, because various remote terminals use different codes for the same characters, and it is the character, not the punch codes that are transmitted.

Card Input Conversion Modes

Card input is reformatted according to the conversion modes specified on the ++FORMAT control card. In all of the discussions, it is assumed that prior to translation a card consists of 80 characters with trailing blanks as required. The action of each translation mode is as follows:

TRIM	strips off trailing blanks (default).
LOWERCASE	converts all uppercase characters to their lowercase equivalent unless preceded by the escape character "\".
ADDNL	appends a newline character after the last character of a card. This operation takes place after trimming, if trimming was requested (default).

CONTIN	if the last character on the card is the escape character "\" then if ADDNL mode is specified, a newline character is not added. This operation takes place after trimming, if trimming was requested.

The opposite of the above modes cause the actions described above not to occur. Thus:

NOTRIM	the trailing blanks of the card image are not removed.
NOCONVERT	no uppercase to lowercase conversion is performed (default).
NOADDNL	a newline character is not appended after the last character.
NOCONTIN	no action is taken if the last character on the card is the escape character "\" (default).

If the user is reading a deck, using edit-directed I/O, into a PL/I program that expects card images to be fixed-length records, the following card input modes should be used:

```
NOCONTIN
NOCONVERT
NOTRIM
ADDNL
```

Deck Size

Decks being read in mcc or rmcc format may exceed the maximum length of a Multics segment. If they do the input is automatically stored in a multisegment file.

Errors

The operator returns a note with the deck if any errors take place during the read. In general, the error should be corrected and the deck resubmitted.

PUNCHED CARD OUTPUT

The card decks produced as a result of the enter_output_request and dpunch commands have some additional punched cards before and after the requested data. These cards are used to separate each deck and to identify the deck and its owner. The identification cards, punched only from the local card punch, are punched with a pattern of holes that can be easily read when the card is flipped over (flip card format).

The complete deck looks like the following:

```
SEPARATOR CARD
Info Cards - punched in flip card format (local card punch only)
SEPARATOR CARD
User's Data - punched in the requested format
END OF DECK - punched in flip card format (local card punch only)
SEPARATOR CARD
```

All cards punched in flip card format and the separator cards must be removed before the deck can be read using the Multics Card Input Facility.

Card-Output Conversion Modes

The Multics Card Code (mcc) conversion mode is best suited to files consisting of ASCII character data. Each character is punched in one card column. When a newline character is encountered in the file, the remainder of the current card is left blank and the following line begins on the next card. Lines longer than 80 characters are punched on several cards. If decks containing such lines are read back into Multics, additional newline characters appear in the file.

The raw conversion mode is suited only for segments that contain complete binary card images. Any checksums, sequence numbers or bit counts to be punched must already be contained in the binary card images. The segment punched must be a multiple of 960 bits long if the deck is to be read back into Multics.

The 7punch conversion mode essentially furnishes a binary representation of any file, suitable for subsequent reloading. The 7punch format also provides sequencing and checksum computation. The format is primarily useful when a file is being punched in order to serve as additional backup and not for use on any system other than Multics. The Multics 7punch format is as follows:

	Columns						
Rows	1	2	3	4	5	6	7 - 72
1-3	7	w	s	c	c	c	d ... d
4-6	w	w	s	c	c	c	d ... d
7-9	w	t	s	c	c	c	d ... d
10-12	5	s	s	c	c	c	d ... d

where:

1. 7 and 5 (octal)
are 7punch format identifying codes.

2. wwww
is the number of data words on the card, if less than 27(8); if greater, it is a replication count and indicates how many times the single data word on the card is to be replicated on reading back in.

3. t
is a last card code. It is 0 on each card of the deck except on the last card, where it is 3. The bit count of the file is punched as the last card for Multics decks.

4. sssss
is the sequence number of the card in the deck, starting from 0.

5. ccccccccccc
is the full word logical checksum of all bits on the card except the checksum itself.

6. dddd ... dddd
are the data words. On the last card, columns 7-9 contain the bit count (fixed binary(35)) and columns 10-72 are 0. Notice that the word count is 0 on the last/bit count card.

PUNCHED-CARD CODES

The card punch codes used with Multics to represent ASCII characters are based on the card punch codes defined for the IBM EBCDIC standard. The correspondence between the EBCDIC and ASCII character sets is defined automatically. The Multics standard card punch code described here is based on the widely available card handling equipment used with IBM System/360 computers. The six characters for which the Multics standard card code differs from the ASCII card code are noted in Table C-3.

The character set used for symbolic source programs and input/output on Multics is the American Standard Code for Information Interchange, X3.4-1968, known as ASCII. See the description of this set in Appendix A, "ASCII Character Set." The character set used for input/output with some devices from a System/360 computer is the International Business Machines (IBM) standard, known as EBCDIC. This set is described in *IBM Systems Reference Library Manual IBM System/360 Principles of Operation*, A22-6821-7.

Although there are 85 graphics in common between EBCDIC and ASCII, there is no practical algorithm by which one can deduce an EBCDIC code value from the ASCII code value or vice versa. There are, however, enough common graphics to define a correspondence between the graphic parts of the two codes, and thereby establish conventions for communication between computers using the codes. A card punch code for ASCII is defined simultaneously. Table C-1 shows this correspondence as used on Multics. The correspondence between ASCII Code Value in column one and ASCII Meaning in column two is firmly defined by the ASCII standard. Similarly, correspondence among Corresponding EBCDIC Meaning in column three, EBCDIC Code Value in column four, and EBCDIC/Multics Punch Code in column five is firmly defined by the IBM standard. This table provides a correspondence between the first two columns on the one hand, and the last three on the other.

The graphic correspondence in Table C-1 is derived as follows: 85 ASCII graphic characters correspond directly with identical EBCDIC graphics. Three ASCII graphics are made to correspond with the three remaining EBCDIC graphics as follows:

ASCII	*EBCDIC*
acute accent	apostrophe
left slant	cent sign
circumflex	negation

Thus all 88 EBCDIC graphics have an equivalent ASCII graphic. The remaining six ASCII graphics, namely:

```
left and right square brackets
left and right braces
grave accent
overline (tilde)
```

have no EBCDIC graphic equivalent. In Table C-1 they are made to correspond to unassigned EBCDIC codes that, nevertheless, have well-defined card punch code equivalents. Where possible, the unassigned EBCDIC codes chosen result in the same punch card representation as in the proposed ASCII standard card code. Thus a majority of the Multics standard card codes do, in fact, agree with the proposed standard.

Table C-1. Correspondence Between ASCII Characters and EBCDIC Characters

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
000	(NUL)	NUL	00	9-12-0-8-1	
001	(SOH)	SOH	01	9-12-1	
002	(STX)	STX	02	9-12-2	
003	(ETX)	ETX	03	9-12-3	
004	(EOT)	EOT	37	9-7	
005	(ENQ)	ENQ	2D	9-0-8-5	
006	(ACK)	ACK	2E	9-0-8-6	
007	BEL	BEL	2F	9-0-8-7	
010	BS	BS	16	9-11-6	
011	HT	HT	05	9-12-5	
012	NL (LF)	NL	15	9-11-5	(Note 1)
013	VT	VT	0B	9-12-8-3	
014	NP (FF)	FF	0C	9-12-8-4	
015	(CR)	CR	0D	9-12-8-5	
016	RRS (SO)	SO	0E	9-12-8-6	
017	BRS (SI)	SI	0F	9-12-8-7	
020	(DLE)	DLE	10	12-11-9-8-1	
021	(DC1)	DC1	11	9-11-1	
022	HLF (DC2)	DC2	12	9-11-2	
023	(DC3)	TM	13	9-11-3	(Note 3)
024	HLR (DC4)	DC4	3C	9-8-4	
025	(NAK)	NAK	3D	9-8-5	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
026	(SYN)	SYN	32	9-2	
027	(ETB)	ETB	26	9-0-6	
030	(CAN)	CAN	18	9-11-8	
031	(EM)	None	19	9-11-8-1	
032	(SUB)	SUB	3F	9-8-7	
033	(ESC)	ESC	27	9-0-7	
034	(FS)	IFS	1C	9-11-8-4	
035	(GS)	IGS	1D	9-11-8-5	
036	(RS)	IRS	1E	9-11-8-6	
037	(US)	IUS	1F	9-11-8-7	
040	Space	Space	40	(No punches)	
041			5A	11-8-2	(Note 1)
042	"	"	7F	8-7	
043	#	#	7B	8-3	
044	$	$	5B	11-8-3	
045	%	%	6C	0-8-4	
046	&	&	50	12	
047		'	7D	8-5	Maps ASCII acute accent into EBCDIC apostrophe
050	(	(	4D	12-8-5	
051	)	)	5D	11-8-5	
052	*	*	5C	11-8-4	
053	+	+	4E	12-8-6	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
054	,	,	6B	0-8-3	
055	-	-	60	11	
056	.	.	4B	12-8-3	
057	/	/	61	0-1	
060	0	0	F0	0	
061	1	1	F1	1	
062	2	2	F2	2	
063	3	3	F3	3	
064	4	4	F4	4	
065	5	5	F5	5	
066	6	6	F6	6	
067	7	7	F7	7	
070	8	8	F8	8	
071	9	9	F9	9	
072	:	:	7A	8-2	
073	;	;	5E	11-8-6	
074	<	<	4C	12-8-4	
075	=	=	7E	8-6	
076	>	>	6E	0-8-6	
077	?	?	6F	0-8-7	
100	@	@	7C	8-4	
101	A	A	C1	12-1	
102	B	B	C2	12-2	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
103	C	C	C3	12-3	
104	D	D	C4	12-4	
105	E	E	C5	12-5	
106	F	F	C6	12-6	
107	G	G	C7	12-7	
110	H	H	C8	12-8	
111	I	I	C9	12-9	
112	J	J	D1	11-1	
113	K	K	D2	11-2	
114	L	L	D3	11-3	
115	M	M	D4	11-4	
116	N	N	D5	11-5	
117	O	O	D6	11-6	
120	P	P	D7	11-7	
121	Q	Q	D8	11-8	
122	R	R	D9	11-9	
123	S	S	E2	0-2	
124	T	T	E3	0-3	
125	U	U	E4	0-4	
126	V	V	E5	0-5	
127	W	W	E6	0-6	
130	X	X	E7	0-7	
131	Y	Y	E8	0-8	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
132	Z	Z	E9	0-9	
133	[	None	8D	12-0-8-5	(Notes 1,2)
134	\	¢	4A	12-8-2	(Note 1)
135	]	None	9D	12-11-8-5	(Notes 1,2)
136	^		5F	11-8-7	Maps ASCII circumflex into EBCDIC negation.
137	_	_	6D	0-8-5	
140	`	None	79	8-1	(Note 2)
141	a	a	81	12-0-1	
142	b	b	82	12-0-2	
143	c	c	83	12-0-3	
144	d	d	84	12-0-4	
145	e	e	85	12-0-5	
146	f	f	86	12-0-6	
147	g	g	87	12-0-7	
150	h	h	88	12-0-8	
151	i	i	89	12-0-9	
152	j	j	91	12-11-1	
153	k	k	92	12-11-2	
154	l	l	93	12-11-3	
155	m	m	94	12-11-4	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
156	n	n	95	12-11-5	
157	o	o	96	12-11-6	
160	p	p	97	12-11-7	
161	q	q	98	12-11-8	
162	r	r	99	12-11-9	
163	s	s	A2	11-0-2	
164	t	t	A3	11-0-3	
165	u	u	A4	11-0-4	
166	v	v	A5	11-0-5	
167	w	w	A6	11-0-6	
170	x	x	A7	11-0-7	
171	y	y	A8	11-0-8	
172	z	z	A9	11-0-9	
173	{	None	C0	12-0	(Note 2)
174	|		4F	12-8-7	(Note 1)
175	}	None	D0	11-0	(Note 2)
176	~	None	A1	11-0-1	(Note 2)
177	PAD(DEL)	DEL	07	12-7-9	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

NOTES

1. In the punched card code proposed for ASCII in the latest proposed ANSI standard card code, a different card code is used for this character.

2. This graphic does not appear in (or map into any graphic that appears in) the EBCDIC set; it is assigned to an otherwise invalid EBCDIC code value/card code combination.

3. In some applications, the ASCII meaning of this control character might not correspond to the EBCDIC meaning of the corresponding control character.

4. Where the Multics meaning of a control character differs from the ASCII meaning, the ASCII meaning is given in parentheses.

Table C-2. Summary of Extensions to EBCDIC to Obtain Multics Standard Codes

ASCII Character	Unassigned EBCDIC Card Code Chosen	
open bracket	12-0-8-5	
left slant	12-8-2	
close bracket	12-11-8-5	
grave accent	8-1	*
open brace	12-0	*
close brace	11-0	*
overline/tilde	11-0-1	*
acute accent	8-5	*
circumflex	11-8-7	*

* Same as the ASCII choice for this graphic.

Table C-3. Summary of Differences Between Multics Standard Card Codes and Proposed ASCII Standard Card Codes

ASCII Character	Multics Standard Card Code	ASCII Standard Card Code
newline	11-9-5	0-9-5
exclamation point	11-8-2	12-8-7
open bracket	12-0-8-5	12-8-2
left slant	12-8-2	0-8-2
close bracket	12-11-8-5	11-8-2
vertical line	12-8-7	12-11

CARD-INPUT ESCAPE POSSIBILITIES

The programmer faced with the problem of representing ASCII data in the EBCDIC environment must make some arbitrary decisions if he needs to obtain graphic representation of these six characters. One appropriate technique is that the suggested invalid code be used wherever EBCDIC code representation is required (e.g., in cards or in core memory), but, when printing readable output, the invalid codes be printed as escapes or overstrikes.

For example, choosing the left slant as an escape character, the following graphic representation may be borrowed from the teletype escape conventions.

ASCII Graphic	*EBCDIC Escape Representation*
left brace	\(
right brace	\)
tilde	\t
grave accent	\'
left bracket	\<
right bracket	\>
left slant	\134

The last escape is required in order to ensure unambiguous meaning of the left slant as an escape character.

Alternatively, a series of overstrike graphics that are more suggestive of the ASCII graphics being represented may also be used. For example;

ASCII Graphic	*EBCDIC Overstrike Representation*
left brace	≠ (left parenthesis over minus sign)
right brace	≠ (right parenthesis over minus sign)
left bracket	≠ (left parenthesis over equals sign)
right bracket	≠ (right parenthesis over equals sign)
grave accent	≠ (apostrophe over minus sign)
tilde	≠ (double quote over negation sign)

The 34 ASCII control characters and 51 EBCDIC control characters match in 33 cases. The remainder have no correspondence that can be expected to work in most cases. As a result, the programmer transforming character data from one environment to another must study the precise meaning of the control codes in the new environment. For example, some EBCDIC control codes might logically transform into ASCII hardware escape sequences for some hardware devices. Other controls might not be imitable in the new environment and might instead be printed with graphic escape sequences, or possibly ignored.

There is currently no provision in Multics for accepting escape sequences in card input other than on control cards as described above under "Punched Card Codes," except in the case of RMCC mode.

RETURNED OUTPUT CONTROL RECORDS

Described below are the control records that are permitted in output files returned by a remote system to Multics for on-line perusal by Multics users.

All characters on a control record are converted to lower case except those immediately following the escape character (backslash, \). For example, \SMITH.\SYS\MAINT is mapped into Smith.SysMaint.

Control record format is:

- Columns one and two contain ++
- A keyword appears starting in column three
- Remainder of the record is free form
- Continuation of control records is not permitted; the entire record must be contained within one punch or printer record.

Name: ++IDENT

This control record identifies the Multics user who is to receive the output file. All records in the output file before the ++IDENT control record are discarded. All three fields of this control record must be specified in the order shown.

Usage

```
++]IDENT <FILE_NAME> <PERSON_ID> <PROJECT_ID>
```

where:

1. FILE_NAME
   is the name used to identify the output file in system pool storage. It should be unique among the user's output file recently received. In the event of name duplications, the system output receiving process appends a numeric component to the end of the supplied name and creates a duplicate segment for FILE_NAME unhess the OVERWRITE control option is specified on the ++CONTROL record.

2. PERSON_ID
   is the registered person name of the owner of this output file. Only this person is able to copy the file from the pool.

3. PROJECT_ID
   is the registered project name of the owner.

NOTES

Multics person and project names normally begin with uppercase letters. Such names must have the escape character before each uppercase letter, since all letters in a control record are mapped to lowercase except those immediately following the escape character (backslash).

Angle brackets (<>) in the "Usage" line indicate information supplied by the user.

Name: ++CONTROL

This control record is used to modify the operation of the output file receiving software. This record is optional.

Usage

```
++CONTROL <CTL_KEYS>
```

where:

1. CTL_KEYS
specifies the operating modes of the software and may be one of the following:

OVERWRITE
specifies that if an output file already exists with the name given on the ++IDENT control record, the old file is to be deleted before the new file is received. The default action is described under the FILE_NAME argument of the ++IDENT control record.

AUTO_QUEUE
specifies that the output file is to be automatically queued for printing or punching locally, whichever is appropriate. The default action is to not queue the file.

REQUEST_TYPE <RQT_NAME>
RQT <RQT_NAME>
specifies use of the RQT_NAME print/punch queue if this output file is automatically printed or punched. RQT_NAME must identify a request type whose generic type is "printer" for print files and "punch" for punch files. (See the description of print_request_types in the Commands manual.) If this ctl_key is not given and automatic queuing is requested, the request type established by the system administrator for output from this remote system will be used. This ctl_key is ignored unless the AUTO_QUEUE ctl_key is also given.

Name: ++FORMAT

This control record is used to specify the conversion modes used to format the data in the output file. This record is optional.

Usage

```
++FORMAT <MODES>
```

where:

1. MODES
may be any of the following modes. The meaning of these modes are discussed earlier in this appendix.

```
TRIM
NOTRIM          (default)
LOWERCASE
NOCONVERT       (default)
ADDNL
NOADDNL         (default)
CONTIN
NOCONTIN        (default)
```

Name: ++INPUT

This control record marks the end of the control records and is required for all output files. The next record is the first record of the user's output file to be placed into system pool storage.

Usage

```
++INPUT
```

There are no fields following the key on this control record.

NOTES

The system treats all records received after the ++INPUT record as data and places them into the output file even if they have control record syntax as described above.

APPENDIX D

STANDARD DATA TYPES

This appendix describes the representation of Multics standard data types. See "Subroutine Calling Sequences" in Appendix H for a discussion of data descriptors. Symbolic names for these types, in the second list below, are also given by std_descriptor_types.incl.pl1.

SUMMARY OF DATA DESCRIPTOR TYPES

```
 1      real fixed-point binary short
 2      real fixed-point binary long
 3      real floating-point binary short
 4      real floating-point binary long
 5      complex fixed-point binary short
 6      complex fixed-point binary long
 7      complex floating-point binary short
 8      complex floating-point binary long
 9      real fixed-point decimal 9-bit
10      real floating-point decimal 9-bit
11      complex fixed-point decimal 9-bit
12      complex floating-point decimal 9-bit
13      pointer
14      offset
15      label
16      entry
17      structure
18      area
19      bit string
20      varying bit string
21      character string
22      varying character string
23      file
24*     label constant (used in symbol tables only)
25*     internal entry constant (used in symbol tables only)
26*     external entry constant (used in symbol tables only)
27*     external procedure (used in symbol tables only)
28      reserved for future use
29      real fixed-point decimal leading overpunched sign 9-bit
30      real fixed-point decimal trailing overpunched sign 9-bit
31-32   reserved for future use
33      real fixed-point binary short unsigned
34      real fixed-point binary long unsigned
35      real fixed-point decimal unsigned 9-bit
36      real fixed-point decimal trailing sign 9-bit
37      reserved for future use
38      real fixed-point decimal unsigned 4-bit
39      real fixed-point decimal trailing sign 4-bit byte-aligned
```

```
40      real fixed-point decimal unsigned 4-bit byte-aligned
41      real fixed-point decimal leading sign 4-bit
42      real floating-point decimal 4-bit
43      real fixed-point decimal leading sign 4-bit byte-aligned
44      real floating-point decimal 4-bit byte-aligned
45      complex fixed-point decimal leading sign 4-bit byte-aligned
46      complex floating-point decimal 4-bit byte-aligned
47      real floating-point hexadecimal short
48      real floating-point hexadecimal long
49      complex floating-point hexadecimal short
50      complex floating-point hexadecimal long
51-57   reserved for future use
58      escape
59      algol68 straight
60      algol68 format
61      algol68 array descriptor
62      algol68 union
63*     picture (used in symbol tables only)
64      pascal typed pointer type
65      pascal char
66      pascal boolean
67      pascal record file type
68      pascal record type
69      pascal set type
70*     pascal enumerated type
71*     pascal enumerated type element
72      pascal enumerated type instance
73*     pascal user defined type
74      pascal user defined type instance
75      pascal text file
76      pascal procedure type
77*     pascal variable formal parameter
78*     pascal value formal parameter
79*     pascal entry formal parameter
80*     pascal parameter procedure
81      real floating-point decimal 9-bit extended 9-bit exponent
82      complex floating-point decimal 9-bit extended 9-bit exponent
83      real floating-point decimal 9-bit generic 36-bit exponent
84      complex floating-point decimal 9-bit generic 36-bit exponent
85      real floating-point binary generic 36-bit exponent
86      complex floating-point binary generic 36-bit exponent
```

Starred types are used in runtime-symbol nodes only and not in argument descriptors.

Symbolic Names for Data Descriptor Types

```
1       real_fix_bin_1_dtype
2       real_fix_bin_2_dtype
3       real_flt_bin_1_dtype
4       real_flt_bin_2_dtype
5       cplx_fix_bin_1_dtype
6       cplx_fix_bin_2_dtype
```

```
 7      cplx_flt_bin_1_dtype
 8      cplx_flt_bin_2_dtype
 9      real_fix_dec_9bit_ls_dtype
10      real_flt_dec_9bit_dtype
11      cplx_fix_dec_9bit_ls_dtype
12      cplx_flt_dec_9bit_dtype
13      pointer_dtype
14      offset_dtype
15      label_dtype
16      entry_dtype
17      structure_dtype
18      area_dtype
19      bit_dtype
20      varying_bit_dtype
21      char_dtype
22      varying_char_dtype
23      file_dtype
24*     label_constant_runtime_dtype
25*     int_entry_runtime_dtype
26*     ext_entry_runtime_dtype
27*     ext_procedure_runtime_dtype
29      real_fix_dec_9bit_ls_overp_dtype
30      real_fix_dec_9bit_ts_overp_dtype
33      real_fix_bin_1_uns_dtype
34      real_fix_bin_2_uns_dtype
35      real_fix_dec_9bit_uns_dtype
36      real_fix_dec_9bit_ts_dtype
38      real_fix_dec_4bit_uns_dtype
39      real_fix_dec_4bit_ts_dtype
40      real_fix_dec_4bit_bytealigned_uns_dtype
41      real_fix_dec_4bit_ls_dtype
42      real_flt_dec_4bit_dtype
43      real_fix_dec_4bit_bytealigned_ls_dtype
44      real_flt_dec_4bit_bytealigned_dtype
45      cplx_fix_dec_4bit_bytealigned_ls_dtype
46      cplx_flt_dic_4bit_bytealigned_dtype
47      real_flt_hex_1_dtype
48      real_flt_hex_2_dtype
49      cplx_flt_hex_1_dtype
50      cplx_flt_hex_2_dtype
59      algol68_straight_dtype
60      algol68_format_dtype
61      algol68_array_descriptor_dtype
62      algol68_union_dtype
63*     picture_runtime_dtype
64      pascal_typed_pointer_type_dtype
65      pascal_char_dtype
66      pascal_boolean_dtype
67      pascal_record_file_type_dtype
68      pascal_record_type_dtype
69      pascal_set_type_dtype
70*     pascal_enumerated_type_dtype
71*     pascal_enumerated_type_element_dtype
72      pascal_enumerated_type_instance_dtype
73*     pascal_user_defined_type_dtype
```

```
74      pascal_user_defined_type_instance_dtype
75      pascal_text_file_dtype
76      pascal_procedure_type_dtype
77*     pascal_variable_formal_parameter_dtype
78*     pascal_value_formal_parameter_dtype
79*     pascal_entry_formal_parameter_dtype
80*     pascal_parameter_procedure_dtype
81      real_flt_dec_extended_dtype
82      cplx_flt_dec_extended_dtype
83      real_flt_dec_generic_dtype
84      cplx_flt_dec_generic_dtype
85      real_flt_bin_generic_dtype
86      cplx_flt_bin_generic_dtype
```

OTHER SYMBOLIC NAMES

```
 1      cobol_comp_6_dtype
 1      cobol_comp_7_dtype
 9      cobol_display_ls_dtype
17      cobol_structure_dtype
21      cobol_char_string_dtype
29      cobol_display_ls_overp_dtype
30      cobol_display_ts_overp_dtype
35      cobol_display_uns_dtype
36      cobol_display_ts_dtype
38      cobol_comp_8_uns_dtype
39      cobol_comp_5_ts_type
40      cobol_comp_5_uns_dtype
41      cobol_comp_8_ls_dtype

 1      ft_integer_dtype
 3      ft_real_dtype
 4      ft_double_dtype
 7      ft_complex_dtype
16      ft_external_dtype
19      ft_logical_dtype
21      ft_char_dtype
47      ft_hex_real_dtype
48      ft_hex_double_dtype
49      ft_hex_complex_dtype
50      ft_hex_complex_double_dtype

 1      algol68_short_int_dtype
 1      algol68_int_dtype
 2      algol68_long_int_dtype
 3      algol68_real_dtype
 4      algol68_long_real_dtype
 7      algol68_compl_dtype
 8      algol68_long_compl_dtype
19      algol68_bits_dtype
19      algol68_bool_dtype
```

```
21      algol68_char_dtype
21      algol68_byte_dtype
22      algol68_struct_struct_char_dtype
20      algol68_struct_struct_bool_dtype

 1      pascal_integer_dtype
 4      pascal_real_dtype
24      pascal_label_dtype
25      pascal_internal_procedure_dtype
26      pascal_exportable_procedure_dtype
27      pascal_imported_procedure_dtype
```

STANDARD DATA TYPE FORMATS

In the following discussion let p be the declared precision of an arithmetic datum. Let n be the declared length of a string datum, and let k be the declared size of an area datum. Figures depicting typical decimal constructions reflect the fact that decimal numbers can consist of up to 59 digits.

Any scaling factor declared for a fixed-point datum is not stored with the datum. The scaling factor is applied to the value of the datum when the value participates in a computation or conversion.

Real Fixed-Point Binary Short (descriptor type 1)

A real, fixed-point, binary, unpacked datum of precision 0<p<36 is represented as a 2's complement, binary integer stored in a 36-bit word.

A real, fixed-point, binary, packed datum of precision 0<p<36 is represented as a 2's complement, binary integer stored in a string of p+1 bits.

In Pascal, there are only three sizes of real, fixed point, binary, packed data. All are represented as 2's complement, binary integers:

An integer datum of precision 0<p<9 is stored in one 9-bit byte and aligned on a byte boundary.

An integer datum of precision 8<p<18 is stored in two 9-bit bytes and aligned on a half word boundary.

An integer datum of precision 17<p<36 is stored in one word and aligned on a word boundary.

Real Fixed-Point Binary Long (descriptor type 2)

A real, fixed-point, binary, unpacked datum of precision 35<p<72 is represented as a 2's complement, binary integer stored in a pair of 36-bit words, the first of which has an even address.

A real, fixed-point, binary, packed datum of precision 35<p<72 is represented as a 2's complement, binary integer stored in a string of p+1 bits.

Real Floating-Point Binary Short (descriptor type 3)

A real, floating-point, binary, unpacked datum of precision $0<p<28$ is represented as a 2's complement, binary fraction m and a 2's complement, binary integer exponent e stored in a 36-bit word.

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $-0.5<m\leq 1$ or $0.5\leq m<1$.

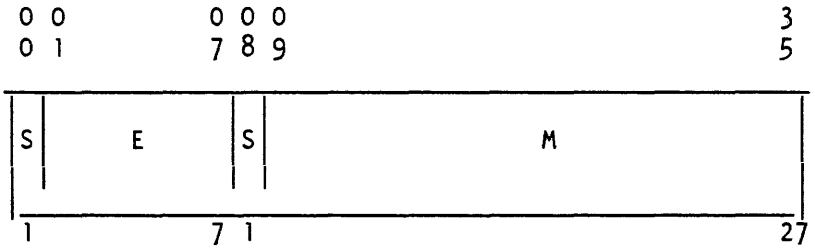


Figure D-1. Single-Precision, Unpacked, Floating-Point Binary-Operand Format

A real, floating-point, binary, packed datum of precision $0<p<28$ is represented as a 2's complement, binary fraction m and a 2's complement, binary integer exponent e stored in a string of p+9 bits.

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $-0.5<m\leq -1$ or $0.5\leq m<1$

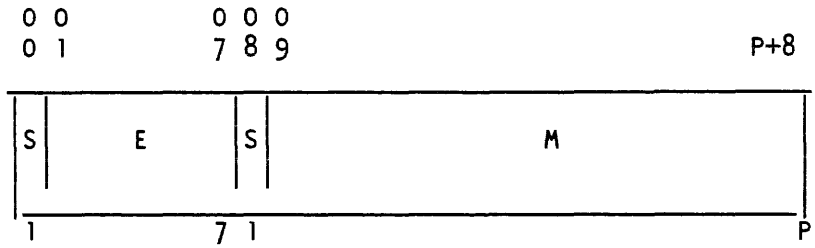


Figure D-2. Single-Precision, Packed, Floating-Point Binary-Operand Format

Real Floating-Point Binary Long (descriptor type 4)

A real, floating-point, binary, unpacked datum of precision $27<p<64$ is represented as a 2's complement, binary fraction m and a 2's complement, binary integer exponent e stored in a pair of 36-bit words, the first of which has an even address.

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $-0.5<m\leq-1$ or $0.5\leq m<1$

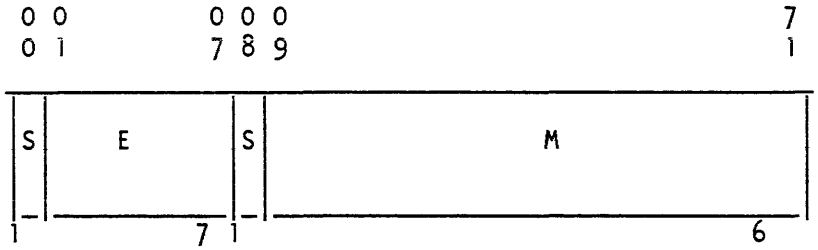


Figure D–3. Double–Precision, Unpacked, Floating–Point Binary–Operand Format

A real, floating–point, binary, packed datum of precision $27<p<64$ is represented as a 2's complement, binary fraction m and a 2's complement, binary integer exponent e stored in a string of p+9 bits.

The value 0 is represented as m=0 and e=-128. For all other values, m satisfies $-0.5<m\leq-1$ or $0.5\leq m<1$

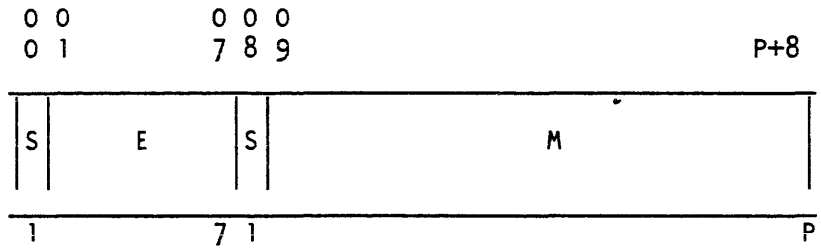


Figure D–4. Double–Precision, Packed, Floating–Point Binary–Operand Format

Complex Fixed–Point Binary Short (descriptor type 5)

A complex, fixed–point, binary, unpacked datum of precision $0<p<36$ is represented as a pair of 2's complement, binary integers stored in a pair of 36–bit words, the first of which has an even address. The first integer is the real part of the complex value and the second integer is the imaginary part of the complex value.

A complex, fixed–point, binary, packed datum of precision $0<p<36$ is represented as a pair of 2's complement, binary integers stored in a string of 2(p+1) bits. The first p+1 bits contain the integer representation of the real part of the complex value and the second p+1 bits contain the integer representation of the imaginary part.

Complex Fixed-Point Binary Long (descriptor type 6)

A complex, fixed-point, binary, unpacked datum of precision 35<p<72 is represented as a pair of 2's complement, binary integers stored in 4 consecutive 36-bit words, the first of which has an even address. The first two words contain the integer representation of the real part of the complex value and the last two words contain the integer representation of the imaginary part.

A complex, fixed-point, binary, packed datum of precision 35<p<72 is represented as a pair of 2's complement, binary integers stored in a string of 2(p+1) bits. The first p+1 bits contain the integer representation of the real part of the complex value and the last p+1 bits contain the integer representation of the imaginary part.

Complex Floating-Point Binary Short (descriptor type 7)

A complex, floating-point, binary, unpacked datum of precision 0<p<28 is represented as a pair of real, floating-point, binary, unpacked data stored in two 36-bit words, the first of which has an even address. The first word contains the real part of the complex value and the second word contains the imaginary part of the complex value.

A complex, floating-point, binary, packed datum of precision 0<p<28 is represented as a pair of real, floating-point, binary, packed data stored in a string of 2(p+9) bits. The first p+9 bits contain the real part of the complex value and the last p+9 bits contain the imaginary part of the complex value.

Complex Floating-Point Binary Long (descriptor type 8)

A complex, floating-point, binary, unpacked datum of precision 27<p<64 is represented as a pair of real, floating-point, binary, unpacked data stored in 4 consecutive 36-bit words, the first of which has an even address. The first two words contain the real part of the complex value and the last two words contain the imaginary part of the complex value.

A complex, floating-point, binary, packed datum of precision 27<p<64 is represented as a pair of real, floating-point, binary, packed data stored in 2(p+9) bits. The first p+9 bits contain the real part of the complex value and the last p+9 bits contain the imaginary part of the complex value.

Real Fixed-Point Decimal Leading Sign 9-bit (descriptor type 9)

A real, fixed-point, decimal, leading sign, 9-bit datum (packed or unpacked) of precision p (where $0<p\leq59$) is represented as a signed, decimal integer stored as a string of p+1 characters. The leftmost character is either a plus (+) or a minus(-), and all other characters are from the set 0123456789.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

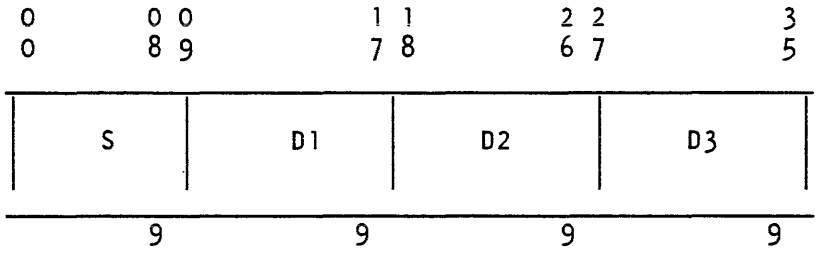


Figure D-5. Typical Type 9 Decimal Datum

Real Floating-Point Decimal 9-bit (descriptor type 10)

A real, floating-point, decimal, 9-bit datum (packed or unpacked) of precision p (where $0<p\leq59$) is represented as a signed, decimal integer m and a 2's complement, binary integer exponent e stored as a string of p+2 characters.

The exponent e is righ justified within the last 9-bit character and the unused bit is zero. The value 0e0 is represented by m=0 and e=+127.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

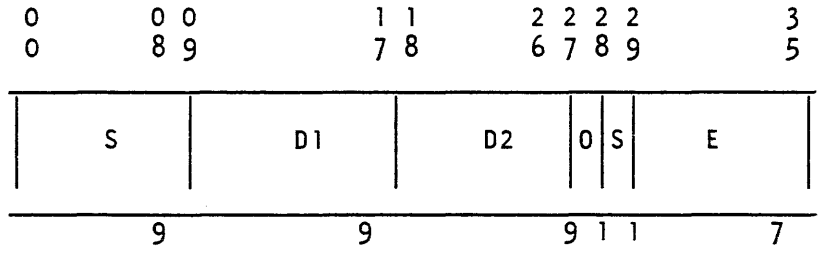


Figure D-6. Typical Type 10 Decimal Datum

Complex Fixed-Point Decimal 9-bit (descriptor type 11)

A complex, fixed-point, decimal datum (packed or unpacked) of precision p is represented as a pair of real, fixed-point, packed, decimal data of precision p. The first datum represents the real part of the complex value, and the second datum represents the imaginary part of the complex value.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

Complex Floating-Point Decimal 9-bit (descriptor type 12)

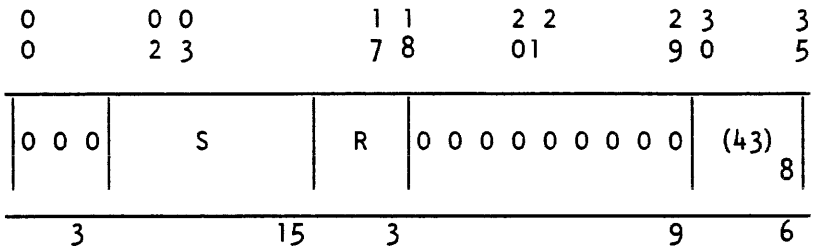
A complex, floating-point, decimal datum (packed or unpacked) of precision p is represented by a pair of real, floating-point, packed, decimal data of precision p. The first datum represents the real part of the complex value and the last represents the imaginary part of the complex value.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

Pointer (descriptor type 13)

An unpacked pointer datum is represented by a ring number r, a segment number s, a word offset w, and a bit offset b, and is stored in a pair of 36-bit words, the first of which has an even address.

Even word of ITS pointer pair:



Odd word of ITS pointer pair:

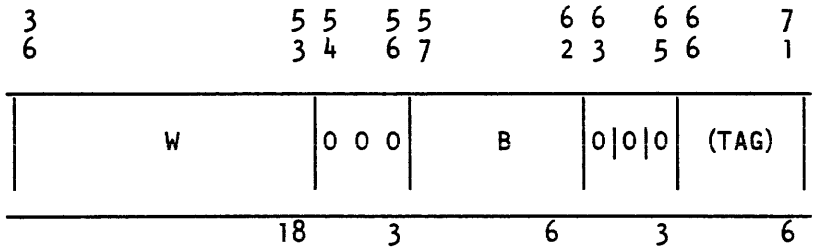


Figure D-7. ITS Pointer Format

A packed pointer datum is represented by a segment number s, a word offset w, and a bit offset b, stored as a string of 36-bits.

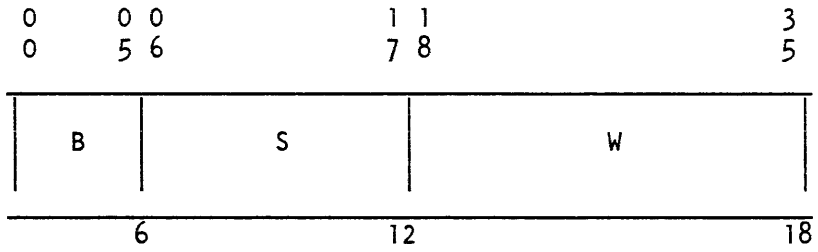


Figure D-8. Packed Pointer Datum Format

Offset (descriptor type 14)

An offset datum (always unpacked) is represented by a word offset w, and a bit offset b, and is stored in a single 36-bit word.

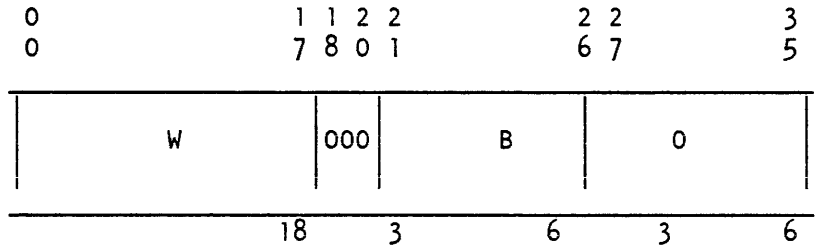


Figure D-9. Offset Datum Format

Label (descriptor type 15)

A label datum (always unpacked) is represented by a pair of unpacked pointers. The first pointer identifies a statement within a procedure and the second pointer identifies a stack frame of an activation of the innermost block containing the statement identified by the first pointer.

Entry (descriptor type 16)

An entry datum (always unpacked) is represented by a pair of unpacked pointers. The first pointer identifies an entry to a procedure and the second identifies a stack frame of an activation of the innermost block containing the procedure whose entry is identified by the first pointer. If the first pointer identifies an entry to an external procedure, the second pointer is null.

Structure (descriptor type 17)

A structure is an ordered sequence of scalar data. A packed structure contains only packed data, whereas an unpacked structure contains either packed or unpacked data or both.

A structure is aligned on a storage boundary that is the most stringent boundary required by any of its components.

An unpacked member of a structure is aligned on a word or double word boundary depending on its data type, and occupies an integral number of words.

A packed member of a structure is aligned on the first unused bit following the previous member, except that up to 8 bits can be unused in order to ensure that a decimal arithmetic or nonvarying string datum is aligned on a 9-bit byte boundary.

An unpacked structure occupies an integral number of words.

Area (descriptor type 18)

An area datum (always unpacked) whose declared size is k occupies k words of storage, the first of which has an even address. The maximum space available for allocations within the area occupies k minus 24 words.

Bit String (descriptor type 19)

A bit string (packed or unpacked) whose length is n occupies n consecutive bits. The leftmost is bit 1 and the rightmost is bit n. An unpacked bit string is aligned on a word boundary and occupies an integral number of words. Some bits of the last word can be unused.

Varying Bit String (descriptor type 20)

A varying bit string (always unpacked) whose maximum length is n is represented by a real, fixed-point, binary, short, unpacked integer m, followed by a nonvarying bit string of length n.

The current length of the varying bit string is the value of m $(0 \leq m \leq n)$. A varying bit string is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bits.

Character String (descriptor type 21)

A character string (packed or unpacked) whose length is n occupies n consecutive 9-bit bytes. When storing 7-bit ASCII characters, each character is right justified within the byte, leaving the two high-order bits to be zero. Characters from the Multics Extended Character Set may also be stored within the byte; in this case at least one high-order bit is nonzero.

An unpacked character string is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

Varying Character String (descriptor type 22)

A varying character string (always unpacked) whose maximum length is n is represented by a real, fixed-point, binary, short, unpacked integer m followed by a nonvarying character string of length n.

The current length of the varying character string is the value of m ($0 \leq m \leq n$).

A varying character string is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

File (descriptor type 23)

A file datum (packed or unpacked) is represented by a pair of unpacked pointers, the second of which points to a file state block and the first of which points to a bit string. Neither the form of the file state block nor the form of the bit string are defined as Multics standards.

Real Fixed-Point Decimal Leading Overpunched Sign 9-bit (descriptor type 29)

A real, fixed-point, decimal, leading overpunched sign, 9-bit datum (packed or unpacked) of precision $0 < p \leq 59$ is represented as a signed decimal integer stored as a string of p characters. The leftmost character represents both the sign and the most significant digit, as shown in Table D-1, and all other characters are chosen from the set 0123456789.

An unpacked datum occupies an integral number of words, the last of which can contain unused bytes.

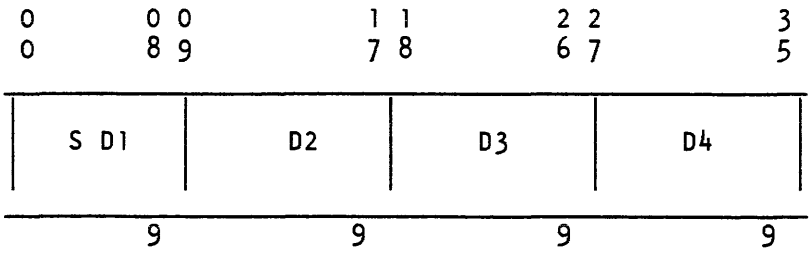


Figure D-10. Typical Type 29 Datum

Table D-1. Overpunched Sign Encoding

Digit	Sign	Octal	ASCII
0	+	173	{
1	+	101	A
2	+	102	B
3	+	103	C
4	+	104	D
5	+	105	E
6	+	106	F
7	+	107	G
8	+	110	H
9	+	111	I
0	-	175	}
1	-	112	J
2	-	113	K
3	-	114	L
4	-	115	M
5	-	116	N
6	-	117	O
7	-	120	P
8	-	121	Q
9	-	122	R

Real Fixed-Point Decimal Trailing Overpunched Sign 9-bit (descriptor type 30)

A real, fixed-point, decimal, leading overpunched sign, 9-bit datum (packed or unpacked) of precision $0<p\leq 59$ is represented as a signed decimal integer stored as a string of p characters. The rightmost character represents both the sign and the most significant digit, as shown in Table D-1, and all other characters are chosen from the set 0123456789.

An unpacked datum occupies an integral number of words, the last of which can contain unused bytes.

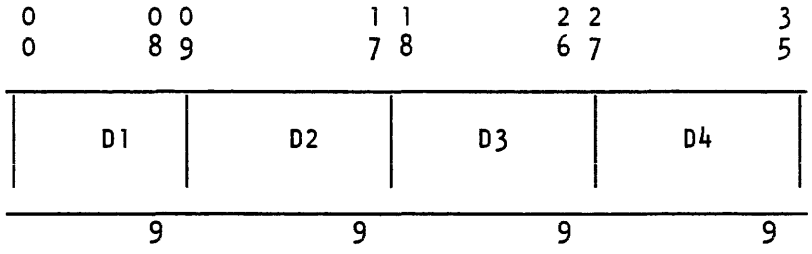


Figure D-11. Typical Type 30 Datum

Real Fixed-Point Binary Short Unsigned (descriptor type 33)

A real, fixed-point, binary, unsigned, unpacked datum of precision $0<p<36$ is represented as a binary integer stored in a 36-bit word.

A real, fixed-point, binary, unsigned, packed datum of precision $0<p<36$ is represented as a binary integer stored in a string of p bits.

Real Fixed-Point Binary Long Unsigned (descriptor type 34)

A real, fixed-point, binary, unsigned, unpacked datum of precision $35<p<72$ is represented as a binary integer stored in a pair of 36-bit words, the first of which has an even address.

A real, fixed-point, binary, unsigned, packed datum of precision $35<p<72$ is represented as a binary integer stored in a string of p bits.

Real Fixed-Point Decimal Unsigned 9-bit (descriptor type 35)

A real, fixed-point, decimal, unsigned, 9-bit datum (packed or unpacked) of precision $0<p\leq59$ is represented as a decimal integer stored as a sequence of p 9-bit bytes.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

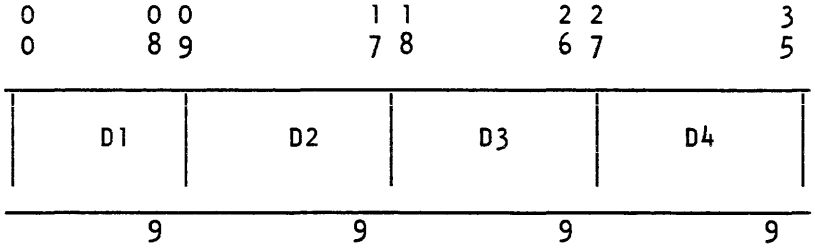


Figure D-12. Typical Type 35 Datum

Real Fixed-Point Decimal Trailing Sign 9-bit (descriptor type 36)

A real, fixed-point, decimal, trailing sign, 9-bit datum (packed or unpacked) of precision $0<p\leq59$ is represented as a signed decimal integer stored as a sequence of p+1 9-bit bytes. The rightmost byte is either a plus sign ("+") or a minus sign ("-") and all other digits are chosen from the set 0123456789.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

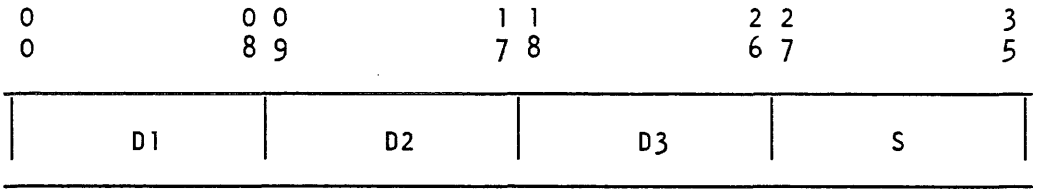


Figure D-13. Typical Type 36 Datum

Real Fixed-Point Decimal Unsigned 4-bit (descriptor type 38)

A real, fixed-point, decimal, unsigned, 4-bit datum (packed or unpacked) of precision $0<p\leq59$ is represented as a decimal integer stored as a sequence of p 4-bit digits.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused digits.

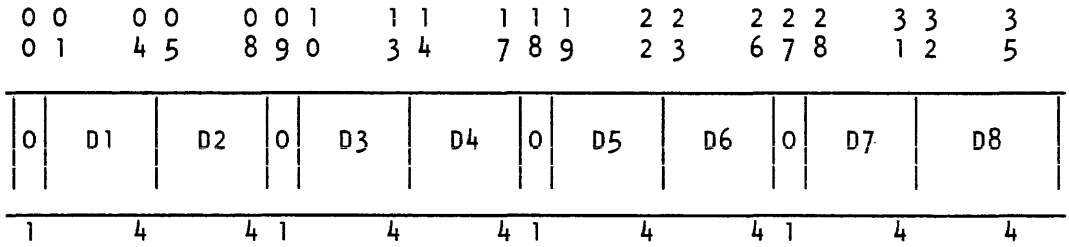


Figure D-14. Typical Type 38 Datum

Real Fixed-Point Decimal Trailing Sign 4-bit Byte Aligned (descriptor type 39)

A real, fixed-point, decimal, trailing sign, 4-bit byte-aligned datum (packed or unpacked) of precision $0<p\leq59$ is represented as a signed decimal integer stored as a sequence of p+1 digits, left justified in a sequence of (p+2)/2 bytes. The rightmost digit is either a plus sign ("1100"b) or minus sign ("1101"b), and all other digits are chosen from the set "0000"b to "1001"b. An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused digits. The last byte can contain an unused digit.

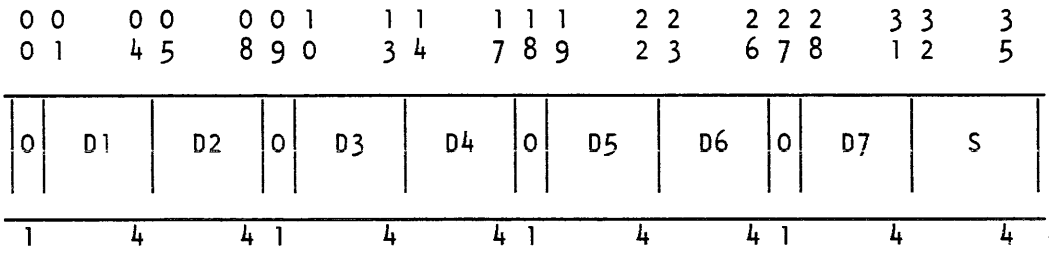


Figure D-15. Typical Type 39 Datum

Real Fixed-Point Decimal Unsigned 4-bit Byte Aligned (descriptor type 40)

A real, fixed-point, decimal, unsigned, 4-bit, byte-aligned datum (packed or unpacked) of precision $0<p\leq59$ is represented as a decimal integer stored as a sequence of p 4-bit digits, left-justified in a sequence of (p+1)/2 bytes. the last byte can contain an unused digit.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused digits.

Real Fixed-Point Decimal Leading Sign 4-bit (descriptor type 41)

A real, fixed-point, decimal, leading sign, 4-bit datum (packed or unpacked) of precision $0<p\leq59$ is represented as a signed decimal integer stored as a sequence of p+1 digits. The leftmost digit is either a plus sign ("1100"b) or a minus sign ("1101"b), and all other digits are chosen from the set "0000"b to "1001"b.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused digits.

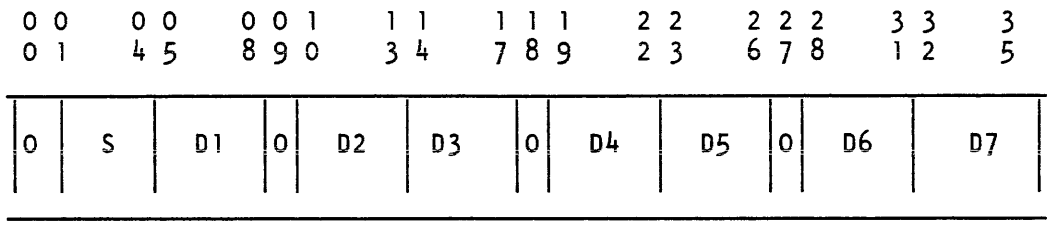


Figure D-16. Typical Type 41 Datum

Real Floating-Point Decimal 4-bit (descriptor type 42)

A real, floating-point, decimal, 4-bit datum (packed or unpacked) of precision $0<p\leq59$ is represented as a signed decimal integer m and a 2's complement, binary integer exponent e stored as a sequence of p+3 digits. The exponent e is stored in two consecutive digits.

The value 0e0 is represented by m=0 and e=+127. An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused digits.

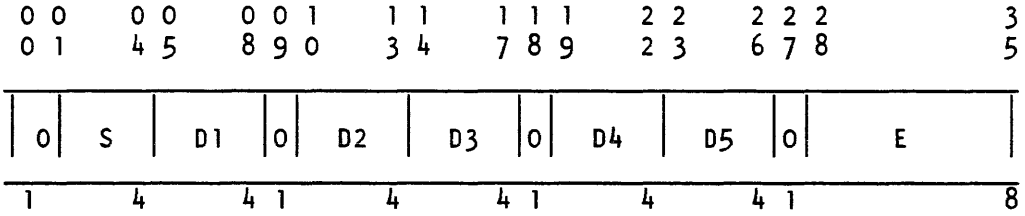


Figure D-17. Typical Type 42 Datum

Real Fixed-Point Decimal Leading Sign 4-bit Byte-Aligned (descriptor type 43)

A real, fixed-point, decimal, leading sign, 4-bit, byte-aligned datum (packed or unpacked) of precision p is represented as a signed decimal integer stored as a sequence of p+1 digits, left-justified in a sequence of (p+2)/2 bytes. The leftmost digit is either a plus sign ("1100b"), or a minus sign ("1101"b), and all other digits are chosen from the set "0000"b to "1001"b. The last byte can contain an unused digit.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused digits.

Real Floating-Point Decimal 4-bit Byte-Aligned (descriptor type 44)

A real, floating-point, decimal, 4-bit, byte-aligned datum (packed or unpacked) of precision $0<p\leq59$ is represented as a signed decimal integer m and a 2's complement, binary integer exponent e stored as a sequence of p+3 digits, left justified in a sequence of (p+4)/2 bytes. The exponent e is stored in two consecutive digits. The last byte can contain an unused digit.

The value 0e0 is represented by m=0 and e=+127.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused digits.

Complex Fixed-Point Decimal Leading Sign 4-bit Byte-Aligned (descriptor type 45)

A complex, fixed-point, decimal, leading sign, 4-bit, byte-aligned datum (packed or unpacked) of precision $0<p\leq59$ is represented as a pair of real, fixed-point, packed, decimal, leading sign, 4-bit, byte-aligned data of precision p. The first datum represents the real part of the complex value, and the second datum represents the imaginary part of the complex value.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

Complex Floating-Point Decimal 4-bit Byte-Aligned (descriptor type 46)

A complex, floating-point, decimal, 4-bit, byte-aligned datum (packed or unpacked) of precision $0<p\leq59$ is represented as a pair of real, floating-point, packed, decimal, 4-bit, byte-aligned data of precision p. The first datum represents the real part of the complex value, and the second datum represents the imaginary part of the complex value.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

Real Floating-Point Hexadecimal Short (descriptor type 47)

A real, floating-point hexadecimal, unpacked datum of precision $0<p<28$ is represented as a 2's complement, hexadecimal normalized fraction m and a 2's complement, binary integer exponent e stored in a 36-bit word. The integer exponent indicates quad bit shifts of the mantissa.

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $-0.0625<m<=1$ or $0.0625<=m<1$.

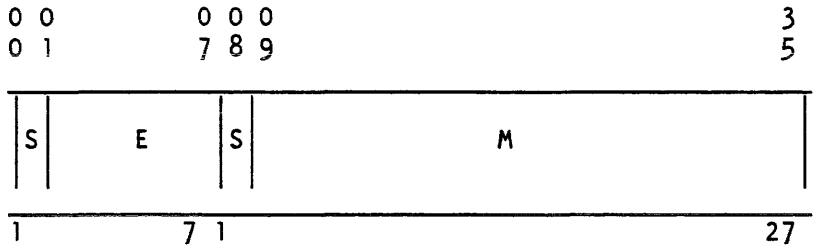


Figure D-18. Single-Precision, Unpacked, Floating-Point Hex-Operand Format

A real, floating-point hexadecimal, packed datum of precision 0<p<28 is represented as a 2's complement, hexadecimal normalized fraction m and a 2's complement, binary integer exponent e stored in a string of p+9 bits. The integer exponent indicates quad bit shifts of the mantissa.

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies -0.0625<m<=1 or 0.0625<=m<1.

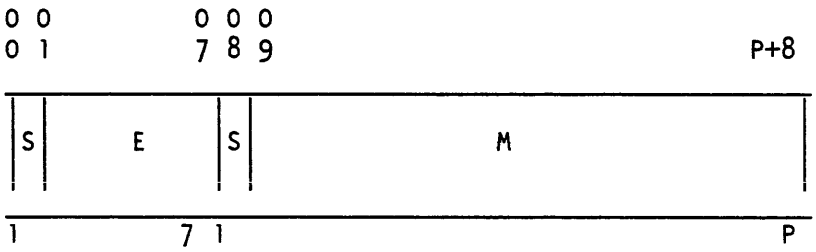


Figure D-19. Single-Precision, Packed, Floating-Point Hex-Operand Format

Real Floating-Point Hexadecimal Long (descriptor type 48)

A real, floating-point hexadecimal, unpacked datum of precision 27<p<64 is represented as a 2's complement, hexadecimal normalized fraction m and a 2's complement, binary integer exponent e stored in a pair of 36-bit words, the first of which has an even address. The integer exponent indicates quad bit shifts of the mantissa.

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies -0.0625<m<=1 or 0.0625<=m<1.

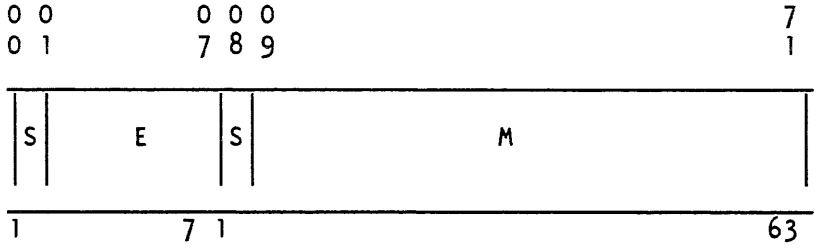


Figure D-20. Double-Precision, Unpacked, Floating-Point Hex-Operand Format

A real, floating-point hexadecimal, packed datum of precision 27<p<64 is represented as a 2's complement, hexadecimal normalized fraction m and a 2's complement, binary integer exponent e stored in a string of p+9 bits. The integer exponent indicates quad bit shifts of the mantissa.

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies -0.0625<m<=1 or 0.0625<=m<1.

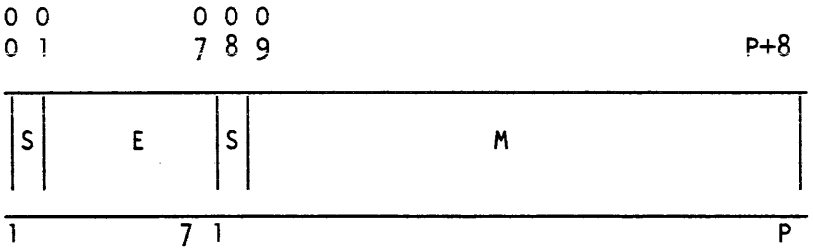


Figure D-21. Double-Precision, Packed, Floating-Point Hex-Operand Format

Complex Floating-Point Hexadecimal Short (descriptor type 49)

A complex, floating-point hexadecimal, unpacked datum of precision 0<p<28 is represented as a pair of real, floating-point, hexadecimal unpacked data stored in two 36-bit words, the first of which has an even address. The first word contains the real part of the complex value and the second word contains the imaginary part of the complex value.

A complex, floating-point hexadecimal, packed datum of precision 0<p<28 is represented as a pair of real, floating-point, hexadecimal unpacked data stored in a string of 2(p+9) bits. The first p+9 bits contain the real part of the complex value and the last p+9 bits contain the imaginary part of the complex value.

Complex Floating-Point Hexadecimal Long (descriptor type 50)

A complex, floating-point hexadecimal, unpacked datum of precision 27<p<64 is represented as a pair of real, floating-point, hexadecimal unpacked data stored in 4 consecutive 36-bit words, the first of which has an even address. The first two words contain the real part of the complex value and the last two words contain the imaginary part of the complex value.

A complex, floating-point hexadecimal, packed datum of precision 27<p<64 is represented as a pair of real, floating-point, hexadecimal unpacked data stored in 2(p+9) bits. The first p+9 bits contain the real part of the complex value and the last p+9 bits contain the imaginary part of the complex value.

Algol68 straight (descriptor type 59)

An Algol68 straight datum (always unpacked) is represented by three pieces of data of different types. The first datum is an entry datum. The second datum is an unpacked pointer. The third datum is a real, fixed-point, binary, short, unpacked integer. An Algol68 straight datum begins on an even address.

The first datum identifies a procedure that returns an element of the straight. The interface of the entry is not defined as a Multics standard. The second datum points to the composite object to be straightened. The third datum is the number of elements in the straight.

Algol68 format (descriptor type 60)

An Algol68 format datum (always unpacked) is represented by three pieces of data of different types. The first datum is an entry datum. The second datum is an unpacked pointer. The third datum is a real, fixed-point, binary, short, unpacked integer. An Algol68 format datum begins on an even address.

The first datum identifies a procedure that interprets the format. The interface of this entry is not defined as a Multics standard. The second datum points to a packed character string. Neither the format of the character string nor the third datum are defined as Multics standards.

Algol68 array descriptor (descriptor type 61)

An Algol68 array descriptor datum is represented by two pieces of data. The first datum is an unpacked pointer. It points to the elements of the array or string. It starts on an even address. The second datum consists of one or more 36-bit words that represent the argument descriptor for the data type of the array. See Appendix H for the representation of an argument descriptor. The number_dims field of the second datum is zero for a VECTOR.

This representation is used when an array occurs as an element of a union, structure, row, or as a REF FLEX [] or REF FLEX VECTOR [] parameter.

Algol68 union (descriptor type 62)

An Algol68 union datum (always unpacked) is represented by three pieces of data. The first datum is an unpacked bit string of length 36. It is unused and reserved for future expansion. The second datum is a real, fixed-point, binary, short, unpacked integer. It represents the ordinal position of the data type of the third datum in the set of constituent data types of the union. The size of the third datum is the largest size of datum in any constituent data type. The format of the third datum is dependent upon its data type. An Algol68 union datum must start on an even address if any constituent data type of the union must start on an even address.

Pascal Typed Pointer Type (descriptor type 64)

A Pascal typed pointer type describes a packed or unpacked pointer which can only be used with data described by the type associated with the pointer. This is enforced by software. A Pascal typed pointer datum actually has the type user-defined type instance.

Pascal Char (descriptor type 65)

A Pascal char datum is a single character with a real, fixed-point, binary, unsigned integer value between 0 and 127 inclusive.

A packed Pascal char datum occupies one 9-bit byte and is aligned on a byte boundary.

An unpacked Pascal char datum occupies the rightmost 9-bit byte of a word that is aligned on a word boundary. The leftmost three bytes of the word are filled with zeroes.

Pascal Boolean (descriptor type 66)

A Pascal boolean datum has an integer value of 0 for FALSE and 1 for TRUE.

A packed Pascal boolean datum occupies one byte and is aligned on a byte boundary.

An unpacked Pascal boolean datum occupies one word and is aligned on a word boundary.

Pascal Record File Type (descriptor type 67)

The Pascal record file type describes a type of file which is an array of records. This type code is used only in runtime symbol tables.

A Pascal record file datum actually has the type Pascal user-defined type instance. It is represented by an unpacked pointer to the Pascal record file status block. The format of a record file status block is not defined as a Multics standard.

Pascal Record Type (descriptor type 68)

The Pascal record type describes a datum which is similar to a structure (descriptor type 17).

An unpacked record is aligned on a double word boundary if it contains fields with double word alignment, such as real, set or pointer. Otherwise it is aligned on a word boundary.

A record that is in a packed record or array has the maximum alignment of any of its fields.

A field of a packed record is usually aligned on the first 9-bit byte boundary that is consistent with the data type's alignment rules. However, if the alignment rules for a field and its successor allow room to unpack the field, the field is unpacked. For example:

```
packed record
  a : integer ;
  b : char ;
  c : integer
end;
```

Fields a and c each occupy one word and are aligned on word boundaries. According to the general rule for packed fields, b should occupy the first byte after a. But since the rest of the word it occupies would be unused, b is in fact unpacked.

Another packing modification is that a field is moved to the rightmost part of the word, if possible. For example:

```
packed record
  a : integer ;
  b,c,d : char ;
  e : integer
end;
```

The integers a and e each occupy one word. Straightforward packing would place b, c and d in the first three bytes after a. However, rather than leaving the rightmost byte of the word unused, the compiler allocates d in the rightmost byte, leaving the third byte unused.

A Pascal record datum actually has the type Pascal user-defined type instance.

Pascal Set Type (descriptor type 69)

The Pascal set type describes a datum which is a nonvarying bit string whose length is equal to the number of elements in the set. The maximum length is 288 bits (8 words). In the following descriptions, L refers to the length in bits.

If $72 < L < 289$, the datum is aligned on a double (even) word boundary and occupies 8 words.

If the set is unpacked and $0 < L < 73$, the datum is aligned on a double (even) word boundary and occupies 2 words.

The alignment rules for packed sets where $L < 73$ are given below:

If $36 < L < 73$, the datum is aligned on a double (even) word boundary and occupies 2 words.

If 18 < L < 37, the datum is aligned on a word boundary and occupies 1 word.

If 9 < L < 19, the datum is aligned on a half-word boundary and occupies 2 9-bit bytes.

If 0 < L < 10, the datum is aligned on a 9-bit byte boundary and occupies one byte.

A Pascal set datum actually has the type Pascal user-defined type instance.

Pascal Enumerated Type (descriptor type 70)

The Pascal enumerated type describes a datum which contains one element of a set of symbolic values. The symbolic values are represented by real, fixed-point, binary, unsigned integers with values from 0 through n-1, where n is the number of elements in the type.

An unpacked datum is aligned on a word boundary and occupies 1 word.

A packed datum is aligned on a half-word boundary, occupying 2 9-bit bytes, when n > 511. A packed datum is aligned on a 9-bit byte boundary, occupying 1 byte, when n <= 511.

This type code is used primarily in runtime symbol tables. It is used in symbol nodes representing the type itself, as in

```
TYPE
enumerated_type = (value1, value2, value3);
```

Data of Pascal enumerated types declared by the program actually have the type Pascal enumerated type instance.

Pascal Enumerated Type Element (descriptor type 71)

A Pascal enumerated type element is one of the symbolic values for a Pascal enumerated type. It is represented by a real, fixed-point, binary, short, unsigned integer.

This type code is used only in runtime symbol tables.

Pascal Enumerated Type Instance (descriptor type 72)

A Pascal enumerated type instance is a datum of a Pascal enumerated type. It is represented by a real, fixed-point, binary, unsigned integer. The alignment and size of the datum depend on the description of the Pascal enumerated type and on the environment where it is declared (a field in a packed record or element of a packed array is packed).

Pascal User Defined Type (descriptor type 73)

A Pascal user-defined type is a type defined by the user either implicitly or explicitly as an array or subrange.

The description of such a type is contained in a runtime symbol table node.

Pascal User Defined Type Instance (descriptor type 74)

A Pascal user-defined type instance is a datum of a Pascal user-defined type.

Pascal Text File (descriptor type 75)

A Pascal text file is represented by an unpacked pointer to a Pascal text file status block. The format of the text file status block is not defined as a Multics standard.

Pascal Procedure Type (descriptor type 76)

The Pascal procedure type type code us used in the runtime symbol table in conjunction with symbol nodes for internal, exported and imported procedures (types 25, 26, 27). Symbol nodes of this generic type are used to anchor descriptions of the parameter lists. They are separated from their associated procedure symbol nodes in order to facilitate comparison of parameter lists.

Pascal Variable Formal Paramater (descriptor type 77)

The Pascal formal variable parameter type code is used in the runtime symbol table to describe a procedure parameter that is passed by reference.

Pascal Value Formal Parameter (descriptor type 78)

The Pascal formal value parameter type code is used in the runtime symbol table to describe a procedure parameter that is passed by value.

Pascal Procedure Formal Parameter (descriptor type 79)

The Pascal formal parameter procedure parameter type code is used in the runtime symbol table to describe a procedure parameter that is itself a procedure or function.

Pascal Procedure Parameter (descriptor type 80)

A Pascal procedure parameter datum (always unpacked is represented by a pair of unpacked pointers followed by a real, fixed-point, binary, short, unpacked integer. The pointers have the same meaning as for an entry datum (type 16). The integer is the offset in the pascal_operators_ transfer vector of the transfer to the call operator to be used for the procedure parameter.

Real Floating-Point Decimal 9-bit Extended 9-bit Exponent (descriptor type 81)

A real, floating-point, decimal, 9-bit datum (packed or unpacked) of precision p (where 0<p<=59) is represented as a signed, decimal integer m and a 2's complement, binary integer exponent e stored as a string of p+2 characters.

The exponent e is a signed 9-bit exponent within the last 9-bit character. The value 0e0 is represented by m=0 and e=+127.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

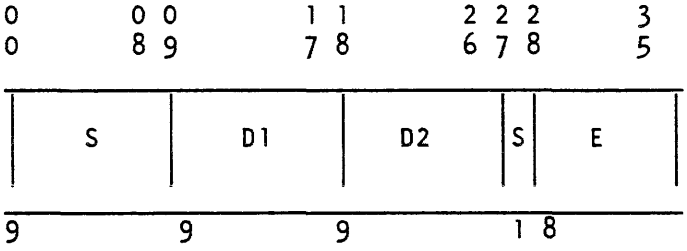


Figure D-22. Typical Type 81 Datum

Complex Floating-Point Decimal 9-bit Extended 9-bit Exponent (descriptor type 82)

A complex, floating-point, extended decimal datum (packed or unpacked of precision p is represented by a pair of real, floating-point, packed, extended decimal data of precision p. The first datum represents the real part of the complex value and the last represents the imaginary part of the complex value.

An unpacked datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

Real Floating-Point Decimal 9-bit Generic 36-bit Exponent (descriptor type 83)

A real, floating-point, decimal, 9-bit unpacked datum of precision p (where 0<p<=59) is represented as a 2's complement 36-bit integer exponent e aligned on a word boundary and a signed, decimal integer m stored as a string of p+5 characters.

The exponent e is a signed 36-bit exponent in the first word. The value 0e0 is represented by m=0 and e=+127.

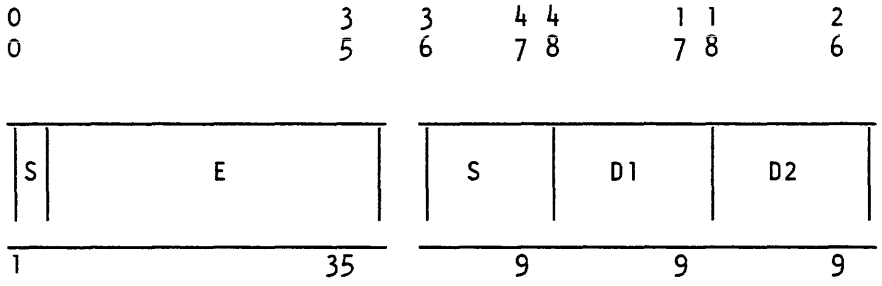


Figure D-23. Typical Type 83 Datum

Complex Floating-Point Decimal 9-bit generic 36-bit Exponent (descriptor type 84)

A complex, floating-point, generic decimal unpacked datum of precision p is represented by a pair of real, floating-point, packed, generic decimal data of precision p. The first datum represents the real part of the complex value and the last represents the imaginary part of the complex value. Any area between the end of the first datum and the exponent word of the second datum is unused.

Real Floating-Point Binary Generic (descriptor type 85)

A real, floating-point, binary, unpacked datum of precision 0<p<64 is represented as a double-precision floating point datum, stored in 3 consecutive words, the first of which has an even address. The exponent part of the double precision floating point datum is a pad field (typically 0) and a 2's complement 36-bit integer exponent e is in the third word of the Generic datum.

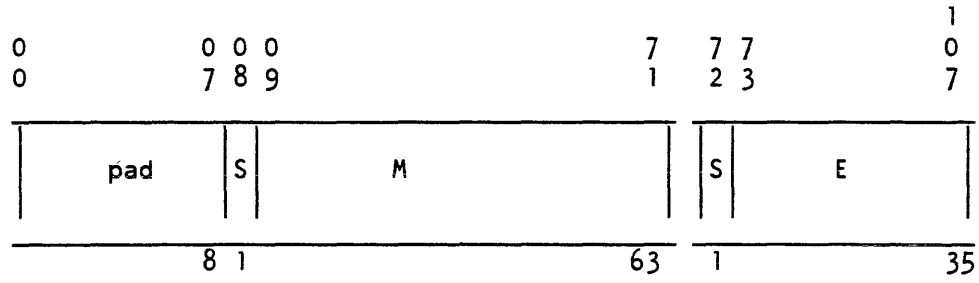


Figure D-24. Floating-Point Binary Generic Format

Complex Floating-Point Binary Generic (descriptor type 86)

A complex, floating-point, binary unpacked generic datum of precision 0<p<64 is represented as a pair of real, floating-point, binary, unpacked generic data stored in 7-consecutive 36-bit words, the first of which has an even address. The first three words contain the real part of the complex value and the last three words contain the imaginary part of the complex value. The fourth word is a pad field.

Arrays

An array is an n-dimensional, ordered collection of scalars or structures, all of which have identical attributes. The elements of an array are stored in row major order (when accessed sequentially the rightmost subscript varies most rapidly) by BASIC, COBOL, and PL/I, and are stored in column major order (when accessed sequentially the leftmost subscript varies most rapidly) by FORTRAN. The user must be aware of such differences when accessing an array with procedures written in two or more languages.

APPENDIX E

LIST OF NAMES WITH SPECIAL MEANINGS

The following names are reserved for special purposes within Multics. The user should not use them with a different meaning.

RESERVED I/O SWITCH NAMES

By convention, the following I/O switch names are reserved. Those switches maintained by the standard environment are:

user_i/o
: is the switch attached to the user's terminal or absentee input and output segments.

user_input
: is the switch attached to user_i/o and devoted expressly to read calls.

user_output
: is the switch attached to user_i/o and devoted expressly to write calls.

error_output
: is the switch attached to user_i/o and devoted expressly to write calls under error conditions.

Those maintained by system commands or subroutines are:

audit_i/o.HHMM.T
: is the switch used by the audit facility, where HHMM.T is a string representing the time the switch was attached.

debug_input
: is the switch attached by the debug command (using the si request).

debug_output
: is the switch attached by the debug command (using the so request).

ec_switch_nn
: is the switch attached by the exec_com command where nn is a unique sequence number assigned by the exec_com command. When the attach control line is used, the switch user_input is attached to this switch through the syn_ I/O module.

fo_uniquename
: is the switch attached by the file_output command. The specified switch is attached to this switch through the syn_ I/O module.

fo_save_uniquename
: is the switch used by the file_output command to save the previous attachment of the specified switch.

filenn
: is the switch attached by the FORTRAN I/O system where nn is the file reference number.

graphic_input
: is the switch used for graphics input.

graphic_output
: is the switch used for graphics output.

uniquename.lila
uniquename.rel
uniquename.res
: are switches used by the LINUS subsystem.

RESERVED SEGMENT NAMES

By convention, the following segment names are reserved. The commands listed that use these sgements are described in the Commands manual. Those segments maintained in the home directory are:

start_up.ec
: is the exec_com invoked at the beginning of a process in the standard environment.

Person_id.breaks
: is the break segment used by the debug command.

Person_id.mbx
: is the mailbox segment used by the print_mail, read_mail, send_mail, and send_message commands.

Person_id.memo
: is the segment used by the memo command.

Person_id.probe
: is the break segment used by the probe command.

Person_id.profile
: is the segment used by the abbrev command.

Person_id.symbols
: is the segment used by the Speedtype commands, add_symbols, use_symbols, and print_symbols_path (described in the Multics WORDPRO Reference Guide, Order No. AZ98).

Person_id.value
: is the default segment maintained by the value facility. The enter_output_request, check_info_segs, and print_motd commands use this segment through the value facility. Users can manipulate values in this segment with the value_get, value_set, value_list, and value_delete commands.

Those maintained in the process directory are:

dseg
: (descriptor segment) is a hardcore ring data segment.

kst
: (known segment table) is a hardcore ring data segment.

pds
: (process data segment) is a hardcore ring data segment.

pit
: is the user's process initialization table. It should only be referenced through the user_info_ subroutine (described in the Subroutines manual).

process_search_segment_.N
: is a segment used to contain search lists for subsystems and commands in ring.N (1<=N<=7).

stack_N
: is the user's automatic storage area for ring number N (1<=N<=7).

unique_name.area.linker
: is a combined linkage segment. There is at least one for each ring in use.

unique_name.temp.N
: are segments maintained by the get_temp_segments_ subroutine, where N is the segment number of the segment.

In general, users should not create segments whose names end in a trailing underscore (_). These names are reserved for system subroutines and may cause errors if they are in the user's search path. (See "Search Rules" in Section 4.)

RESERVED SEGMENT-NAME SUFFIXES

Suffixes are used as in the following example: when creating a PL/I source program to be named xyz, the user would create a source-language segment named xyz.pl1. The PL/I compiler, by convention, translates this segment, producing the segment xyz.list, containing a printable listing, and the segment xyz, containing the object program.

By convention, the following segment-name suffixes are reserved. The language-translator source-segment suffixes are:

Language Translator	*Source Segment*	*Include Files*
PL/I compiler	pl1	incl.pl1
FORTRAN compiler	fortran	incl.fortran
ALM assembler	alm	incl.alm
BASIC compiler	basic	
COBOL compiler	cobol	incl.cobol
ALGOL68 compiler	algol68	incl.algol68
PASCAL compiler	pascal	incl.pascal

The listing segment suffix is:

list
: is the suffix for printed output listing segments produced by compilers, the assembler, and the binder. This suffix is also used on segments created by the -output_file control argument of process_list command when a pathname has not been given explicitly with the control argument.

Other special suffixes include:

absin
: is the input segment suffix for an absentee process.

absout
: is the default output segment suffix for an absentee process.

archive
: is the suffix on the segment created by the archive command.

bind
: is the suffix on the input control segment for the binder.

cds
: is the suffix on a cds source segment.

cdt
: is the suffix on the segment created by cv_cmf.

chars
: is the suffix of the special-purpose segment used by the compose command for comments, indices, etc.

cmdb
: is the suffix on the source segment used by the create_mrds_db command.

cmdsm
: is the suffix on the source segment used by the create_mrds_dsm command.

cmf
: is the suffix on the source segment converted by cv_cmf.

code
: is the suffix on the enciphered segment created by the encode command.

compin
: is the suffix on the segment to be formatted by the compose command.

compout
: is the suffix on the output file formatted by the compose command.

db
: is the suffix on a data-base directory.

dict
: is the suffix on any dictionary segment the user creates.

dir_info
: is the suffix on a segment created by the save_dir_info command.

dsm
: is the suffix on a data submodel.

ec
: is the suffix on the input segment to the exec_com command.

fdocin
: is the suffix on the input segment used by the format_document command.

fdocout
: is the suffix on the output segment created by the format_document command.

gcos
: is the suffix on a segment that is in GCOS standard system format.

gct
: is the suffix on a segment reserved for a graphic character table.

gdt
: is the suffix on a segment reserved for a graphic device table.

graphics
: is the suffix on an output file that contains graphic code.

info
: is the suffix on a segment formatted for use with the help command.

linus
: is the suffix used for LINUS macros.

lister
: is the suffix on the segment created by the create_list command from a listin segment.

listform
: is the suffix on the segment that defines the format of a document produced from a list.

listin
: is the suffix on any segment used to input and update a list used by the create_list command.

mail
: is the suffix on segments generated by the read_mail and send_mail "write" request.

map355
: is the suffix on an FNP source program to be assembled.

mbx
: is the suffix on any mailbox segment the user creates.

memo
: is the suffix on any memo segment the user creates.

ms
: is the suffix on an administrative ring message segment.

pdt
: is the sufix on the segment created by cv_pmf.

pfd
: is the suffix on profile data files output by the profile command.

pfl
: is the suffix on profile listing files output by the profile command.

pmf
: is the suffix on the source segment converted by cv_pmf.

probe
: is the suffix on the segment used by the probe command.

profile
: is the suffix on any profile segment the user creates. A profile segment contains abbreviations used with the abbrev command.

qedx
: is the suffix of a segment containing qedx instructions.

rd
: is the suffix on a translator source segment that is input to the reduction_compiler command.

rdmec
: is the suffix on the input segment to the send_mail "exec_com" request.

rtdt
: is the suffix on the segment converted by cv_rtmf.

rtmf
: is the suffix on the source segment converted by cv_rtmf.

runoff
: is the input segment suffix to the runoff command.

runout
: is the output segment suffix from the runoff command.

sat
: is the suffix on the segment created by cv_smf.

sdmec
: is the suffix on the input segment to the send_mail "exec_com" request.

smf
: is the suffix on the source segment converted by cv_smf.

sv.apl
: is the suffix on the segment containing a saved workspace from the apl command.

symbols
: is the suffix of the symbol dictionary used by Speedtype commands.

table
: is the suffix on a segment created from a data model or submodel by the create_mrds_dm_table command.

ttf
: is the suffix on source segment converted by cv_ttf.

ttt
: is the suffix on the segment created by cv_ttf.

value
: is the suffix on data bases maintained by value_set and related commands.

volumes
: is the suffix of a segment processed by the manage_volume_pool command.

wl
: is the suffix on any wordlist segment the user creates.

RESERVED OBJECT-SEGMENT ENTRY POINT

By convention, the following entry-point definition in object segments is reserved.

symbol_table
: is the entry-point definition that provides the address of the symbol table produced by the compilers.

Since this is a reserved entry point, no user-created program can use this name. A statement of the form:

```
symbol_table:  procedure...
```

is illegal if it is the external procedure block.

APPENDIX F

MULTICS STANDARD MAGNETIC TAPE FORMAT

This appendix describes the standard physical format used on 7-track and 9-track magnetic tapes on Multics. Tapes of this form may be written and read by the tape_mult_ I/O module (described in the Subroutines manual). Any magnetic tape not written in the standard format described here is not a Multics standard tape.

STANDARD TAPE FORMAT

The first recorded block on the tape following the beginning of tape (BOT) mark is the tape label record. Following the tape label record is an end of file (EOF) mark. Subsequent reels of a multireel sequence also have a tape label followed by an EOF mark. (An EOF mark is the standard sequence of bits on a tape that is recognized as an EOF by the hardware.)

Following the tape label and its associated EOF are the data records. An EOF is written after every 128 data records with the objective of increasing the reliability and efficiency of reading and positioning within a logical tape. Records that are repeated because of transmission, parity, or other data alerts are not included in the count of 128 records. The first record following the EOF has a physical record count of 0 mod 128.

An end of reel (EOR) sequence is written at the end of recorded data. An EOR sequence is:

```
EOF mark
EOR record
EOF mark
EOF mark
```

STANDARD RECORD FORMAT

Each physical tape block consists of a modulo 4096 character (1024 word) data space enclosed by an 8-word (32 character) header and an 8-word trailer. The total block length is then the total data space (expressed in characters) plus 64 characters of overhead. For a block with a data space of 4096 characters plus overhead (4160 characters total), the following table illustrates the physical space requirements for each data block and its associated inter-record gap.

tracks	density	block distance	gap distance
7	800	7.800 inches	.75 inches
9	800	5.850 inches	.60 inches
9	1600	2.925 inches	.50 inches
9	6250	0.688 inches	.30 inches

For 6250 bpi 9-track tape, use the attach description control argument "-density 6250" or "-den 6250."

PHYSICAL RECORD HEADER

The following is the format of the physical record header:

Word 0
: Constant with octal representation 670314355245.

Words 1 and 2
: Multics standard unique identifier (70 bits, left justified). Each record has a different unique identifier.

Word 3
: Bits 0-17: the number of this physical record in this physical record, beginning with record 0.
Bits 18-35: the number of this physical file on this physical reel, beginning with file 0.

Word 4
: Bits 0-17: the number of data bits in the data space, not including padding.
Bits 18-35: the total number of bits in the data space.
If word 5 bit position 23 is a one bit, then these two integers are interpreted as being the number of data characters used in the data space and the total number of characters contained in the data space respectively.

Word 5
: Flags indicating the type of record. Bits are assigned considering the leftmost bit to be bit 0 and the rightmost bit to be bit 35. Word 5 also contains a count of the rewrite attempt, if any.

Bit	*Meaning if Bit is 1*
0	This is an administrative record (one of bits 1 through 13 is 1).
1	This is a label record.
2	This is an end of reel (EOR) record.
3-13	Reserved.
14	One or more of bits 15-26 are set.
15	This record is a rewritten record.
16	This record contains padding.
17	This record was written following a hardware end of tape (EOT) condition.
18	This record was written synchronously; that is, control did not return to the caller until the record was written out.
19	The logical tape continues on another reel (defined only for an end of reel record).
20-22	Reserved.
23	The data bits used and data bit length integers in header word 4 are interpreted as characters used and character length respectively.
24-26	Header version number, currently 2.
27-35	If bits 14 and 15 are 1, this quantity indicates the number of the attempt to rewrite this record. If bit 15 is 0, this quantity must be 0.

Word 6

Contains the checksum of the header and trailer excluding word 6; i.e., excluding the checksum word. (See below for a description of standard checksum computation.)

Word 7

Constant with octal representation 512556146073.

PHYSICAL RECORD TRAILER

The following is the format of the trailer:

Word 0
Constant with octal representation 107463422532.

Words 1 and 2
Standard Multics unique identifier (duplicate of header).

Word 3
Total cumulative number of data bits for this logical tape (not including padding and administrative records).

Word 4
Padding bit pattern (described below).

Word 5
Bits 0-11: reel sequence number (multireel number), beginning with reel 0.

Bits 12-35: physical file number, beginning with physical file 0 of reel 0.

Word 6
The number of the physical record for this logical tape, beginning with record 0.

Word 7
Constant with octal representation 265221631704.

NOTE: The octal constants listed above were chosen to form elements of a single error-correcting code whether read as 8-bit tape characters (9-track tape) or as 6-bit tape characters (7-track tape).

ADMINISTRATIVE RECORDS

The standard tape format includes three types of administrative records: a standard tape label record, a bootable tape label record, and an end of reel (EOR) record.

Standard Tape Label Record

The standard tape label record is written in standard record format and can best be defined by the PL1 structure declaration that follows:

```
dcl 1 stand_label_record          based (mstrp) aligned,
      2 head                      like mstr_header,
      2 installation_id           char (32),
      2 tape_reel_id              char (32),
      2 volume_set_id             char (32),
      2 pad (1000)                bit (36),
      2 trail                     like mstr_trailer;
```

where:

1. head
   is the standard 8-word record header described above.

2. installation_id
   is the ASCII installation code. This identifies the installation that labeled the tape.

3. tape_reel_id
   is the ASCII reel identification. This is the reel identification by which the operator stores and retrieves the tape.

4. volume_set_id
   is the name of the volume set if the "-volume_set_name" tape_mult_ attach description argument was used when the tape reel was created. If the "-volume_set_name" attach description argument was not used, this field is padded with ASCII blanks.

5. pad
   is an array of words containing the standard padding pattern (described below), used to fill the label record data space out to the standard size.

6. trail
   is the standard 8-word record trailer described above.

Bootable Tape Label Record

The bootable tape label record is an administrative record, written in nonstandard format. The first eight words of the physical record contain four pairs of executable instructions collectively known as a transfer vector. The function of this transfer vector is to allow a Multics standard tape to be bootloaded from any of four possible I/O controllers.

When a tape that contains a bootable tape label record is bootloaded, a hardwired program within the I/O controller writes the data within the first record starting at location 30 (octal, absolute) in memory. When the data transfer is completed, the I/O controller sets an interrupt "cell" in the system controller, which causes the bootload processor to execute a hardwired "XED" instruction to the address indicated by the system controller. This interrupt address generated by the system controller is a function of the interrupt "cell" set by the I/O controller and by the configuration panel number of the I/O controller itself. For example, if the bootload sequence was initiated on I/O controller #0, then the interrupt address would be 30 (8); addresses 32, 34, and 36, respectively, would be generated by I/O controllers number 1, 2 and 3. The executable instructions contained in each pair of the transfer vector are:

```
lda     4
tra     330
```

Location 4 contains the DCW address stored by the I/O controller's hardwired boot program. An executable program is located at 330 (octal, absolute). This program is known as the tape label boot program.

The bootable tape label record is created through the use of the tape_mult_ control order "boot_program". This control order is normally executed by the generate_mst command to write a bootable label on BOS system tapes. Although a user may write his own boot program and have generate_mst write it out to the BOS tape label, a standard boot label exists in the system libraries with the name of mst_boot_label.

The function of the mst_boot_label boot program is to initialize the bootstrap environment and set up an I/O channel program to read and skip the EOF record, and to read in the first data record on the tape under control of a DCW. The DCW address used is 7750 (8) absolute with a word count of 4096. (The generate_mst command places the standard 8-word tape record header plus a 16-word segment header before the first data in the record; the first executable data in the record starts at location 10000 (8).) After the first data record is read in, the status returned from the tape controller is checked for errors. If an error occurs, the status word is copied in the A register and the processor falls into a DIS. Assuming no status error is detected, control is transferred to absolute location 10000 (8).

There are many other fields in the standard tape label record. The following is a PL1 structure definition of the contents of the tape label record followed by an explanation of each field:

```
dcl 1 mst_label            based (mstrp) aligned,
      2 xfer_vector        (4),
        3 lda_instr        bit (36),
        3 tra_instr        bit (36),
      2 head               like mstr_header,
      2 installation_id    char (32) unaligned,
      2 tape_reel_id       char (32) unaligned,
      2 volume_set_id      char (32)
      2 fv_overlay         char (32) analigned,
        3 scu_instr        bit (36),
        3 dis_instr        bit (36),
      2 fault_data (8)     bit (36),
      2 boot_pgm_path      char (168) unaligned,
      2 userid char        (32) unaligned
      2 label_version      fixed bin,
      2 output_mode        fixed bin,
      2 boot_pgm_len       fixed bin,
      2 copyright          char (56) unaligned,
      2 pad (13)           bit (36),
      2 boot_pgm           (0 refer (mst_label.boot_pgm_len))
                                bit (36),
      2 trail              like mstr_trailer;
```

where:

1. xfer_vector
   is the bootload transfer vector. There is one transfer vector for each of four possible I/O controllers. The transfer vector functions to gain control as the result of an interrupt after a bootload sequence.

2. lda_instr
   is an "LDA" instruction from absolute location 4, which for an IOM is the payload channel DCW as stored by the hardwired bootload program in the IOM.

3. tra_instr
   is an unconditional transfer to the beginning of the bootload program.

4. head
   is the standard 8-word record header described above.

5. installation_id
   is the ASCII installation code. This identifies the installation that labeled the tape.

6. tape_reel_id
is the ASCII reel identification. This is the reel identification by which the operator stores and retrieves the tape.

7. volume_set_id
is the name of the volume set if the "-volume_set_name" tape_mult_ attach description argument is used when the tape reel is created. If the "-volume_set_name" attach description argument is not used, this field is padded with ASCII blanks.

8. fv_overlay
This 32-element array overlays the hardware fault vector area at absolute location 100 (octal) if this tape is bootloaded. If an unexpected fault occurs when this tape is bootloaded, the appropriate fault pair is executed by the processor fault logic.

9. scu_instr
is a Store Control Unit (SCU) instruction, which safe-stores the state of the processor control unit when executed.

10. dis_instr
is an interrupt inhibited Delay until Interrupt Signal (DIS) instruction, which halts the processor when executed.

11. fault_data
is an area where SCU data is stored if an unexpected fault occurs while bootloading this tape.

12. boot_pgm_path
if nonblank, it can be the absolute pathname of the boot program written on this label record. It can also be the user-designated name for the boot program when the "boot_program" tape_mult_ control order is executed.

13. userid
is the User_id (Person.Project.Instance) of the user who created this tape.

14. label_version
is the version number of this label record structure, currently 2.

15. output_mode
is the number of the iox_ mode in effect when this tape was created. (See iox_modes.incl.pl1.)

16. boot_pgm_len
is the length of the boot program in words. The boot program must be less than or equal to 840 (1510 octal) words in length. If it is less than 840 words, the record is padded out with the standard padding pattern.

17. boot_pgm
is the executable text of the boot program. The boot program must be coded in absolute self-relocating ALM assembly language.

18. trail
is the standard 8-word record trailer described above.

End of Reel Record

The end of reel record contains only padding bits in its data space. The standard record header of the EOR record contains the information that identifies it as an EOR record (word 5, bits 0 and 2 are one).

DENSITY AND PARITY

Both 9-track and 7-track standard tapes are recorded in binary mode with odd ones having lateral parity. Standard densities are 800 frames per inch (bpi) (recorded in NRZI mode), 1600 bpi (recorded in PE mode), and 6250 bpi (recorded in GCR mode).

DATA PADDING

The padding bit pattern is used to fill administrative records and the last data record of a reel sequence.

COMPATIBILITY CONSIDERATION

Software shall be capable of reading Multics Standard tapes that are written with records with less than 1024 words in their data space. In particular, a previous Multics standard tape format specified a 256-word (9216-bit) data space in a tape record.

In addition to recognizing and reading standard and bootable tape label records, software shall also be capable of recognizing and reading Multics standard tapes that were generated with a version 1 label record, that is, standard label records that do not contain the volume_set_id field.

STANDARD CHECKSUM

The checksum described below is the standard Multics technique for computing a full word checksum on the Honeywell DPS 8/M computer.

Algorithm

Checksums are computed using the "awca" instruction followed by an "alr 1" instruction. Upon completion of checksum computation, two "awca 0,dl" instructions are executed to include all carries in the checksum.

A typical checksum computation scheme follows:

```
        sti     indices         save indicators
        lda     0,dl            initialize "a" to zero
        eaxl    0               count locations in xl

 loop:  ldi     indics          restore indicators
        awca    word,l          add with carry to checksum
        sti     indics          save indicators (they get
                                clobbered by cmpxl)
        alr     l               rotate "a" left
        eaxl    l,l             count l location and
        cmpxl   size,du         check for completion
        tnc     loop            loop

        ldi     indics          restore indicators
        awca    0,dl            add in carry, if any
        awca    0,dl            in case carry generated by
                                last instruction
        sta     cksum           save the checksum
```

APPENDIX G

MULTICS STANDARD OBJECT FILE WITH SYMBOL TABLE ORGANIZATION

A multics object file is either an object segment or an object multisegment-file (MSF). Object MSFs are created by the linkage editor (le) and consist of a number of standard object segments.

FORMAT OF AN OBJECT MULTISEGMENT FILE

An object multisegment file consists of a number of object segments in bound object segment format. All executable code is stored in components 1 through n, where n is the largest components number in the MSF. Component 0 is generated by the linkage editor and contains no text section. The definition section of component 0 contains an indirect definition for each of the visible definitions from each of the other components. The linkage section of component 0 contains a partially snapped link to the target of each indirect definition, a normal link to the procedure msf_prelink_, a *link link to the base of the linkage section, and a first-reference trap which calls the msf_prelink_ procedure to reaolve all of the partially snapped links in the MSF.

FORMAT OF AN OBJECT SEGMENT

A Multics object segment contains object code generated by a translator and linkage information that is used by the dynamic linking mechanism to resolve intersegment references. (See "Dynamic Linking" in Section 4.) The most common examples of object segments are procedure segments and data segments.

Format requirements for an object segment are primarily associated with external interfaces; thus, translator designers are permitted a great amount of freedom in the area of code and data generation. The format contains certain redundancies and unusual data structures; these are a byproduct of maintaining upward compatibility with earlier object segment formats. The dynamic linking mechanism and the standard object segment manipulation tools assume that all object segments are standard object segments.

An object segment is divided into six sections that usually appear in the following order:

```
text
definition
linkage
static (if present)
symbol
break map (if present)
```

The type of information contained in each of the six sections is summarized below:

1. text
   text contains only pure parts of the object segment (instructions and read-only data). It can also contain relative pointers to the definition, linkage and symbol sections.

2. definition
   contains only nonexecutable, read-only symbolic information used for dynamic linking and symbolic debugging. Since it is assumed that the definition section is infrequently referenced (as opposed to the constantly referenced text section), it should not be used as a repository for read-only constants referenced during the execution of the text section. The definition section can sometimes (as in the case of an object segment generated by the binder) be structured into definition blocks that are threaded together.

3. linkage
   contains the impure (i.e., modified during the program's execution) nonexecutable parts of the object segment and may consist of two types of data:

   a. links modified at run time by the Multics linker to contain the machine address of external references, and possibly

   b. data items to be allocated on a per-process basis such as the internal static storage of PL/I procedures.

4. static
   contains the data items to be allocated on a per-process basis. The static storage may be included in the linkage section in which case there is no explicit separate static section.

5. symbol
   contains all generated items of information that do not belong in the first five sections such as the language processor's symbol tree and historical and relocation information. The symbol section may be further structured into variable length symbol blocks threaded to form a list. The symbol section contains only pure information.

6. break map
   contains information used by the debuggers to locate breakpoints in the object segment. This section is generated by the debuggers rather than the translator and only when the segment currently contains breakpoints. Its internal format is of interest only to the debuggers.

The text, definition, and symbol sections are shared by all processes that reference an object segment. Usually, a copy of the linkage section is made when an object segment is first referenced in a process. That is, the linkage section is a per-process data base. The original linkage section serves only as a copying template. An exception is made for some system programs whose link addresses are filled in at system initialization time. Their linkage sections are shared by everyone who wants to use the supplied addresses. When these programs have data items in internal storage, they have a separate static section template that is copied once per process. See Section 4 and "Standard Stack and Linkage Area Formats" in Appendix H of this document. Normally, a segment containing break map information is in the state of being debugged and is not used by more than one process.

The object segment also contains an object map that contains the offsets and lengths of each of the sections. The object map can be located immediately before or immediately after any of the six sections. Translators normally place it immediately after the symbol section. The last word of every object segment must contain a left-justified 18-bit relative pointer to the object map.

STRUCTURE OF THE TEXT SECTION

The text section is basically unstructured, containing the machine-language representation of a symbolic algorithm and/or pure data. Its length is usually an even number of words.

Two of the items that can appear within the text section have standard formats: the entry sequence and the gate segment entry point transfer vector.

Entry Sequence

A standard entry sequence is usually provided for every externally accessible procedure entry point in an object segment. A standard entry sequence has the following format, defined by the system include files entry_sequence_info.incl.pl1 and entry_desc_info.incl.pl1:

```
dcl 1 parm_desc_ptrs              aligned based,
      2 n_args                    fixed bin(18) unsigned unaligned,
      2 descriptor_relp           (num_descs refer
                                    (parm_desc_ptrs.n_args))
                                  bit(18) unaligned;

dcl num_descs                     fixed bin(18);

dcl 1 entry_sequence              aligned based,
      2 word1,
        3 descr_relp_offset       bit(18) unaligned,
        3 reserved                bit(18) unaligned,
      2 word2,
        3 def_relp                bit(18) unaligned,
        3 flags                   unaligned like entry_desc_info.flags,
      2 code_sequence             bit(36) aligned;

dcl 1 entry_desc_info             aligned based(entry_desc_info_ptr),
      2 version                   fixed bin,
      2 flags,
        (3 basic_indicator,
        3 revision_1,
        3 has_descriptors,
        3 variable,
        3 function)               bit(1) unaligned,
        3 pad                     bit(13) unaligned,
      2 object_ptr                ptr,
      2 bit_count                 fixed bin(24);

dcl entry_desc_info_versin_2      fixed bin int static options
                                    (constant) init(2),
    entry_desc_info_ptr           ptr;
```

STRUCTURE ELEMENTS

n_args
is the number of arguments expected by this external entry point. This item is optional and is valid only if the flag has_descriptors equals "1"b.

descriptor_relp
is an array of pointers (relative to the base of the text section) to the descriptors of the corresponding entry point parameters. This item is optional and is valid only if the flag has_descriptors equals "1"b. See "Parameter Descriptors" in Appendix H.

descr_relp_offset
is the offset (relative to the base of the text section) of the n_args item. This item is optional and is valid only if the flag has_descriptors equals "1"b.

reserved
is reserved for future use and must be "0"b.

def_relp
is an offset (relative to the base of the definition section) to the definition of this entry point. Thus, given a pointer to an entry point, it is possible to reconstruct its symbolic name for purposes such as diagnostics or debugging.

flags
contains binary indicators that provide information about this entry point.

basic_indicator
"1"b this is the entry point of a BASIC program
"0"b this is not the entry point of a BASIC program

revision_1
"1"b all of the entry's parameter descriptor information is with the entry sequence, i.e., none is in the definition
"0"b parameter descriptor information, if any, is with the definition

has_descriptors
"1"b the entry has parameter descriptors; i.e., items n_args, descriptor_relp and descr_relp_offset contain valid information
"0"b the entry does not have parameter descriptors

variable
"1"b the entry expects arguments whose number and types are variable
"0"b the number and type of arguments, if any, are not variable

function
"1"b the last parameter is to be returned by this entry
"0"b the last parameter is not to be returned by this entry

pad is reserved for future use and must be "0"b

code_sequence
is any sequence of machine instructions satisfying Multics standard calling conventions. See "Subroutine Calling Sequences" in Appendix H.

function
is on if the procedure entry point is a function which returns a value. The final parameter argument descriptor describes this return value.

object_ptr
if the entry descriptor is being taken from an archive, this is the pointer to the base of the archive component. (Output) Otherwise, this is null.

bit_count
if the entry descriptor is being taken from an archive, this is the bit count of the archive component. (Output) Otherwise, this is zero.

entry_desc_info_version_2
is a named constant which the caller should use to set the version number in the structure above.

The value (i.e., offset within the text section) of the entry point corresponds to the address of the code_sequence item. (The value is stored in the formal definition of the entry point. See "Structure of the Definition" below.) Thus, if entry_offset is the value of the entry point ent1, then the def_relp item pointing to the definition for ent1 is located at word (entry_offset minus 1).

Gate Segment Entry Point Transfer Vector

For protection purposes, control must not be passed to a gate procedure at other than its defined entry points. To enforce this restriction, the first n words of a gate segment with n entry points must be an entry point transfer vector. That is, the kth word $(0 \leq k \leq n-1)$ must be a transfer instruction to the kth entry point (i.e., a transfer to the code_sequence item of a standard entry sequence as described above). In this case, the value of the kth entry point is the offset of the kth transfer instruction

(i.e., word k of the segment) rather than the offset of the code_sequence item of the kth entry point.

To ensure that only these entries can be used, the hardware enforced entry bound of the gate segment must be set so that the segment can be entered only at the first n locations.

STRUCTURE OF THE DEFINITION SECTION

The definition section of an object segment contains pure information that is used by the dynamic linking mechanism.

The definition section consists of a header pointing to a linked list of items describing the externally accessible named items of the object segment, followed by an unstructured area containing information describing the externally accessible named items of other object segments referenced by this object segment. The linked list is known as the definition list. The items on the list are known as definitions. The unstructured area contains expression words, type pairs, trap words, trap procedure information, and the symbolic names associated with external references.

All structures for dealing with the definition section are contained in the system include file in definition_dcls.incl.pl1.

A definition specifies the name of an externally accessible named item and its location in the object segment. The definition list consists of one or more definition blocks each of which consists of one or more class-3 definitions followed by zero or more definitions that are not class-3 (see "Definition Section Header" below for format). Normally, unbound object segments contain one definition block, while bound segments contain one definition block for every component object segment.

Optionally, the definition section can contain a definition hash table. If present, the hash table is used by the linker to expedite the search for a definition.

The information in the unstructured area of the definition section is used at runtime in conjunction with information in the linkage section to resolve the external references made by the object segment. This information is conceptually part of the linkage section, but is stored in the definition section so it can be shared among all the users of the segment.

Figure G-1 shows the structure of the definition section. For more information concerning the interpretation of the information in the definition section see "Dynamic Linking" in Section 4.

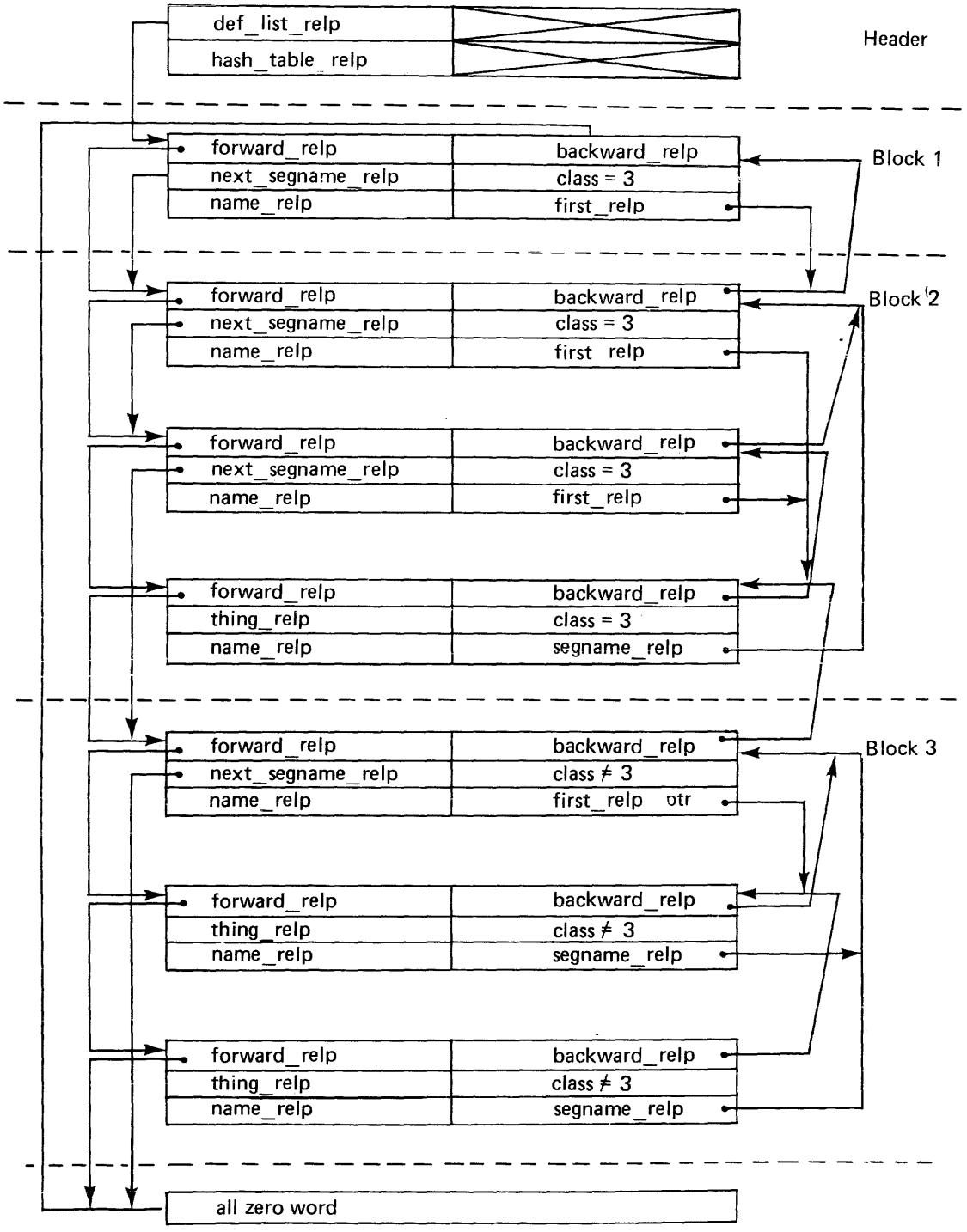


Figure G-1. Sample Definition List

Character strings in the definition section are stored in ALM "acc" format. This format is described by the following PL/I declaration, defined by the system include file acc.incl.pl1:

```
dcl 1 acc          based aligned,
      2 num_chars  fixed bin(9) unsigned unaligned,
      2 string     char(0 refer(acc.num_chars)) unaligned;
```

The first nine bits of the string contain the length of the string. Unused bits of the last word of the string must be zero. Such a structure is referred to as an acc string. Another structure declaration (acc_string) exists in definition_dcls.incl.pl1 for easier allocation of the string and zeroing of the unused bits at the end of the string.

The following paragraphs describe the formats of the various items in the definition section.

Definition Section Header

The definition section header resides at the base of the definition section and contains an offset (relative to the base of the definition section) to the beginning of the definition list (in definition_dcls.incl.pl1).

```
dcl 1 definition_header  aligned based,
      2 def_list_relp    fixed bin(18) unsigned unaligned,
      2 msf_map_relp     bit(18) unsigned unaligned,
      2 hash_table_relp  fixed bin(18) unsigned unaligned,
      2 flags            unaligned,
        3 new            bit(1) unaligned initial ("1"b),
        3 ignore         bit(1) unaligned initial ("1"b),
        3 unused         bit(16) unaligned;
```

STRUCTURE ELEMENTS

def_list_relp
: is a relative pointer to the first definition in the definition list.

msf_map_relp
: is a relative pointer to the msf_map. If no msf_map is present (i.e., the object segment is not a component of an object msf) this pointer must be 0.

hash_table_relp
: is a relative pointer to the beginning of the definition hash table. If no definition hash table is present, this pointer must be "0"b.

flags
: contains binary indicators that provide information about this definition section:

  new
  "1"b definition section has new format
  "0"b definition section has old format

ignore
"1"b if new equals "1"b, the Multics linker ignores this definition.
"0"b is an old format definition

unused is reserved for future use and must be "0"b

A definition that is not class-3 has the following format (in definition_dcls.incl.pl1):

```
dcl 1 definition                        aligned based,
      2 forward_relp                    fixed bin(18) unaligned unsigned,
      2 backward_relp                   fixed bin(18) unaligned unsigned,
      2 thing_relp                      fixed bin(18) unaligned unsigned,
      2 flags                           unaligned,
        3 new                           bit (1) unaligned,
        3 ignore                        bit (1) unaligned,
        3 entry                         bit (1) unaligned,
        3 retain                        bit (1) unaligned,
        3 argcount                      bit (1) unaligned,
        3 descriptors                   bit (1) unaligned,
        3 unused                        bit (8) unaligned,
      2 class                           fixed bin(3) unaligned unsigned,
      2 name_relp                       fixed bin(18) unaligned unsigned,
      2 segname_relp                    fixed bin(18) unaligned unsigned,
      2 n_args                          bit(18) unaligned,
      2 descriptor_relp(0 refer(n_args)) bit(18) unaligned;
```

STRUCTURE ELEMENTS

forward_relp
is a thread (relative to the base of the definition section) to the next definition. The thread terminates when it points to a word that is 0. This thread provides a single sequential list of all the definitions within the definition section.

backward_relp
is a thread (relative to the base of the definition section) to the preceding definition.

thing_relp
is the offset, within the section designated by the class variable (described below), of this symbolic definition.

flags
contains binary indicators that provide additional information about this definition:

new
"1"b definition section has new format
"0"b definition section has old format

ignore
"1"b definition does not represent an external symbol and is, therefore, ignored by the Multics linker
"0"b definition represents an external symbol

entry
"1"b definition of an entry point (a variable reference through a transfer of control instruction)
"0"b definition of an external symbol that does not represent a standard entry point

retain
"1"b definition must be retained in the object segment (by the binder)
"0"b definition can be deleted from the object segment (by the binder)

argcount (obsolete)
"1"b definition includes a count of the argument descriptors (i.e., item n_args below contains valid information)
"0"b no argument descriptor information is associated with the definition

descriptors (obsolete)
"1"b definition includes an array of argument descriptor (i.e., items n_args and descriptor_relp below contain valid information)
"0"b no valid descriptors exist in the definition

indirect
"1"b definition refers to a partial link to the actual target of the symbolic definition which resides in another MSF component
"0"b definition refers directly to the target of this symbolic definition

unused is reserved for future use and must be "0"b

class
this field contains a code indicating the section of the object segment to which value is relative. Codes are:

```
0   text section
1   linkage section
2   symbol section
3   this symbol is a segment name
4   static section
```

name_relp
is an offset (relative to the base of the definition section) to an aligned acc string representing the definition's symbolic name.

segname_relp
is an offset (relative to the base of the definition section) to the first class-3 definition of this definition block.

n_args (obsolete)
is the number of arguments expected by this external entry point. This item is present only if argcount or has_descriptors equals "1"b. This item is not defined in the system include file.

descriptor_relp (obsolete)
is an array of pointers (relative to the base of the text section) that point to the descriptors of the corresponding entry point arguments. This item is present only if has_descriptors equals "1"b. This item is not defined in the system include file.

The obsolete items are described here to illustrate earlier versions; translators should put these items in the entry sequence of the text section. See "Entry Sequence" above.

A class-3 defintition has the following format (in definition_dcls.incl.pl1):

```
dcl 1 segname_definition  aligned based,
      2 forward_relp      fixed bin(18) unaligned unsigned,
      2 backward_relp     fixed bin(18) unaligned unsigned,
      2 next_segname_relp fixed bin(18) unaligned unsigned,
      2 flags             bit(15) unaligned,
      2 class             fixed bin(3) unaligned unsigned,
      2 name_relp         fixed bin(18) unaligned unsigned,
      2 first_relp        fixed bin(18) unaligned unsigned,
```

STRUCTURE ELEMENTS

forward_relp
is the same as above.

backward_relp
is the same as above.

next_segname_relp
is a thread (relative to the base of the definition section) to the next class-3 definition. The thread terminates when it points to a word that contains all 0's. This thread provides a single sequential list of all class-3 definitions in the object segment.

flags
is the same as above.

class
is the same as above (and has a value of 3).

name_relp
is the same as above.

first_relp
is an offset (relative to the base of the definition section) to the first nonclass-3 definition of the definition block. If the block contains no nonclass-3 definitions, it points to the first class-3 definition of the next block. If there is no next block, it points to a word that is all 0's.

The end of a definition block is determined by one of the following conditions (whichever comes first):

- forward_relp points to an all zero word;
- the current entry's class is not 3, and forward_relp points to a class-3 definition;

- the current definition is class 3, and both forward_relp and first_relp point to the same class-3 definition.

The threading of definition entries is shown in Figure G-1 above. The following paragraphs describe items in the unstructured portion of the definition section.

MSF MAP

The MSF map is pointed to by the msf_map_relp pointer in the definition header. It identifies an object segment as a component of an object multisegment file, and indicates the extents of the file. The msf map has the following format (in definition_dcls.incl.pl1):

```
dcl 1 msf_map              aligned based,
      2 version            char (8),
      2 component_count    fixed bin,
      2 my_component       fixed bin;
```

STRUCTURE ELEMENTS

version
is the version of the structure.

component_count
is the number of components in the object multisegment file.

my_component is the number (in the range 0 to component_count-1) of this component.

Expression Word

The expression word is the item pointed to by the expression pointer of an unsnapped link (see "Structure of the Linkage Section" below) and has the following format (in definition_dcls.incl.pl1):

```
dcl 1 exp_word          aligned based,
      2 type_relp       fixed bin(18) unaligned unsigned,
      2 expression      fixed bin(17) unaligned;
```

STRUCTURE ELEMENTS

type_relp
is an offset (relative to the base of the definition section) to the link's type pair.

expression
is a signed value to be added to the offset (i.e., offset within a segment) of the resolved link.

Type Pair

The type pair defines the external symbol pointed to by a link and has the following format (in definition_dcls.incl.pl1):

```
dcl 1 type_pair            aligned based,
      2 type               fixed bin(18) unaligned unsigned,
      2 trap_relp          fixed bin(18) unaligned unsigned,
      2 segname_relp       fixed bin(18) unaligned unsigned,
      2 offset_name_relp   fixed bin(18) unaligned unsigned;
```

STRUCTURE ELEMENTS

type
assumes a value from 1 to 6:

1 is a self-referencing link (i.e., the segment in which the external symbol is located is the object segment containing this link or a dynamic related section of the link) of the form:

```
myself|0+expression,modifier
```

2 unused; it was earlier used to define a now obsolete ITP-type link.

3 is a link referencing a specified reference name but no symbolic offset name, of the form:

```
refname|0+expression,modifier
```

4 is a link referencing both a symbolic reference name and a symbolic offset name, of the form:

```
refname|offsetname+expression,modifier
```

5 is a self-referencing link having a symbolic offset name, of the form:

```
myself|offsetname+expression,modifier
```

6 (obsolete)
same as type 4 except that the external item is created if it is not found.

trap_relp
is an offset (relative to the base of the definition section) to either an initialization structure (if type equals 5 and seg_ptr equals 5, or if type equals 6) or to a trap word.

segname_relp
is a code or a offset depending on the value of type. For types 1 and 5, this item is a code that can assume one of the following values, designating the sections of the self-referencing object segment:

0 is a self-reference to the object's text section; such a reference is represented symbolically as "*text".

1 is a self-reference to the object's linkage section; such a reference is represented symbolically as "*link".

2 is a self-reference to the object's symbol section; such a reference is represented symbolically as "*symbol".

4 is a self-reference to the object's static section; such a reference is represented symbolically as "*static".

5 is a reference to an external variable managed by the linker; such a reference is represented symbolically as "*system".

For types 3, 4, and 6, this item is an offset (relative to the base of the definition section) to an aligned acc string containing the reference name portion of an external reference.

offset_name_relp
has a meaning depending on the value of type. For types 1 and 3, this value ignored and must be zero. For types 4, 5, and 6, this item is an offset (relative to the base of the definition section) to an aligned acc string containing the entry point name of an external reference. If type equals 5 and seg_ptr equals 5, the acc string contains the name of the external variable. (See Section 3 for a discussion of entry point names.)

This page intentionally left blank.

Trap Word

The trap word is a structure that specifies a trap procedure to be called before the link associated with the trap word is resolved by the dynamic linking mechanism. It consists of relative pointers to two links. (Links are defined under "Structure of the Linkage Section" below.) The first link defines the entry point in the trap procedure to be called. The second link defines a block of information that is passed as one of the arguments of the trap procedure. The trap word has the following format (in definition_dcls.incl.pl1):

```
dcl 1 link_trap_pair  aligned based,
      2 call_relp     fixed bin(18) unaligned unsigned,
      2 info_relp     fixed bin(18) unaligned unsigned;
```

STRUCTURE ELEMENTS

call_relp
is an offset (relative to the base of the linkage section) to a link defining the entry point of the trap procedure.

info_relp
is an offset (relative to the base of the linkage section) to a link defining information of interest to the trap procedure.

Initialization Structure for Type 5 System and Type 6 Links

This structure specifies how a link target first referenced because of a type 5 *system or a type 6 link should be initialized. It has the following format (in definition_dcls.incl.pl1; other versions exist in system_link_init_info.incl.pl1):

```
dcl 1 link_init_copy_info  aligned based,
      2 n_words            fixed bin,
      2 type               fixed bin,
      2 initial_data       (0 refer (link_init_copy_info.n_words))
                             bit(36) aligned;
```

STRUCTURE ELEMENTS

n_words
is the number of words required by the new variable.

type
indicates what type of initialization is to be performed. It can have one of the following values:

0 no initialization is to be performed

3 copy the info array into the newly defined variable

4 initialize the variable as an area

initial_data
is the image to be copied into the new variable. It exists only if code is 3.

Definition Hash Table

A definition hash table may be present in the definition section of an object segment. In its basic form, the definition hash table contains an array of pointers to definitions. The definition hashing algorithm selects a particular pointer. If the selected pointer does not point to the desired definition, a linear search is then performed until the appropriate definition is found or a zero pointer is encountered. The initial hash code is generated by taking the remainder of the first word of the definition name (the count and first three characters of the "acc" format string) divided by the size of the hash table. The hash table size is such that it is never more than 80% full.

In bound segments, different components may contain definitions with identical names. In this case, a second hash table is required in order to resolve ambiguities. In addition to this second hash table, a duplicate name table must be provided for each duplicated definition name.

The format of the tables described above is shown in Figure G-2 and is described below:

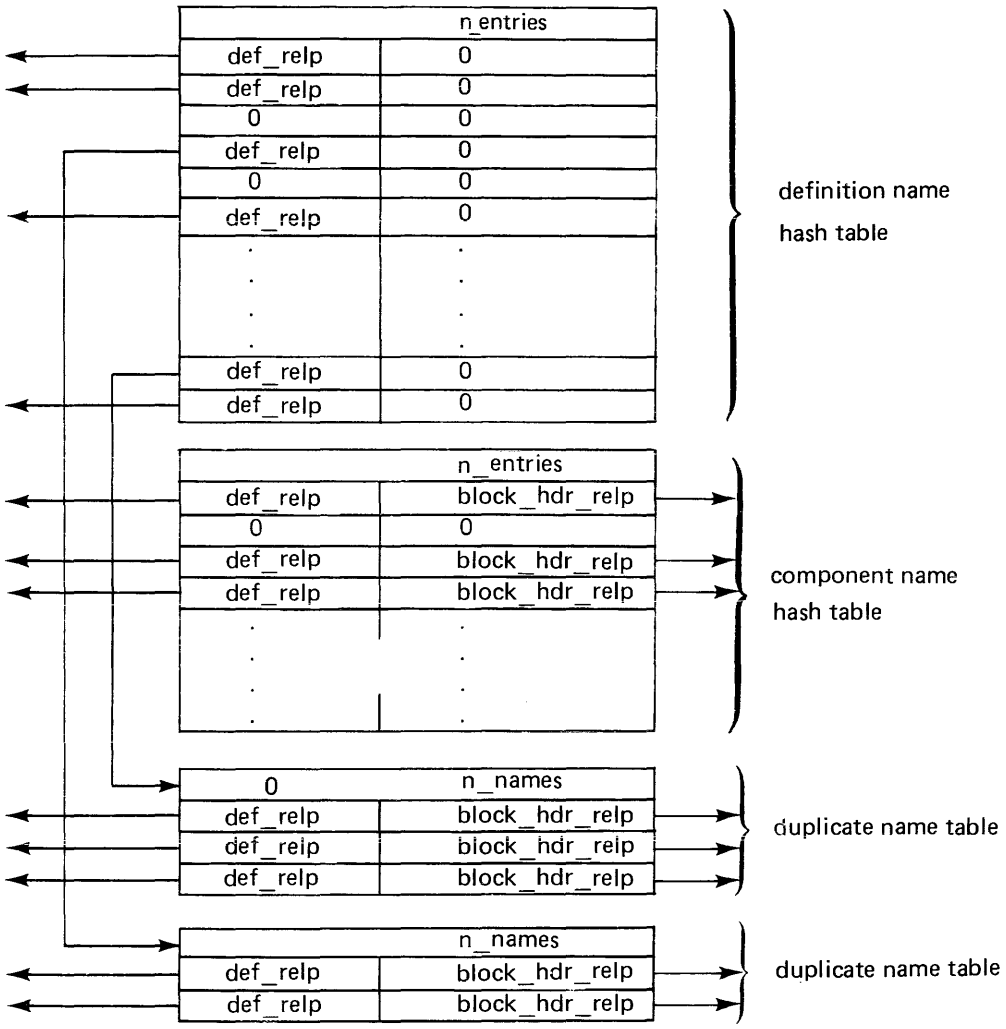


Figure G-2. Definition Hash Table

The definition name hash table is pointed to by a relative pointer in the definition section header (in definition_dcls.incl.pl1). It must contain one nonzero entry for each nonclass-3 definition name.

```
dcl 1 definition_ht     aligned based,
      2 n_entries       fixed bin,
      2 table           (0 refer (definition_ht.n_entries)),
        (3 def_relp     fixed bin(18) unsigned,
         3 unused       bit(18)) unal;
```

STRUCTURE ELEMENTS

n_entries
is the number of elements in the hash table.

def_relp
is an array of pointers to nonclass-3 definitions. In the case of a duplicated definition name, a particular def_relp does not point directly to a definition, but rather to a duplicate name table (see below).

A component name hash table is present only if duplicated definition names are present in a bound segment (in definition_dcls.incl.pl1). It must immediately follow the definition hash table. There is one entry in this hash table for each bound segment component name and synonym (i.e., for each class-3 definition).

```
dcl 1 component_ht          aligned based,
      2 n_entries           fixed bin,
      2 table               (0 refer (component_ht.n_entries)),
        (3 def_relp         fixed bin(18) unsigned,
         3 block_hdr_relp   bit(18)) unaligned;
```

STRUCTURE ELEMENTS

n_entries
is the number of elements in the component name hash table.

table
contains one nonzero element for each class-3 definition.

def_relp
is a relative pointer to a class-3 definition.

block_hdr_relp
is a relative pointer to the first class-3 definition of the definition block containing the definition pointed to by defp.

A duplicate name table must be supplied for each duplicated definition name (in definition_dcls.incl.pl1). Each table has one entry for each instance of the duplicated name. The definition searching algorithm can determine whether the relative pointer retrieved from the definition hash table points to a definition or to a duplicate name table by examining the left half of the first word pointed to. A definition never contains a zero forward_relp, while a duplicate name table is never nonzero in the left half of the first word.

```
dcl 1 duplicate_table              aligned based,
      2 mbz                        bit(18) unaligned,
      2 n_duplicate_table_names    fixed bin unaligned,
      2 table                      (0 refer (duplicate_table.n_names)),
        (3 def_relp                fixed bin(18),
         3 block_hdr_help          fixed bin(18)) unaligned;
```

STRUCTURE ELEMENTS

mbz
must be zero to distinguish from a definition.

n_duplicate_table_names
is the number of instances of a given duplicated name.

table
contains one element for each instance of the duplicated name.

def_relp
is a pointer to a nonclass-3 definition.

block_hdr_relp
is a pointer to the first class-3 definition of the definition block containing the nonclass-3 definition.

Definition searching with a definition hash table is done by first searching for the definition name. If no duplicate name table is encountered, no ambiguity exists and the correct definition is quickly found. If a duplicate name table is encountered, the component name hash table must be searched. Then, a linear search is done on the duplicate name table to match a block_hdr_relp with the block_hdr_relp in the component name hash table.

STRUCTURE OF THE STATIC SECTION

The static section is unstructured.

STRUCTURE OF THE LINKAGE SECTION

The linkage section is subdivided into four distinct components:

1. A fixed-length header that always resides at the base of the linkage section
2. A variable length area used for internal (static) storage (optional)
3. A variable length structure of links (optional)
4. First-reference trap (optional)

These four components are located within the linkage section in the following sequence:

```
header
internal storage (if present)
links (if present)
trap (if present)
```

The length of the linkage section must be an even number of words and must start on an even-word boundary; in addition, the link substructure must also begin at an even location (offset) within the linkage section.

When an object segment is first referenced in a process, its linkage section is copied into a per-process data base. At this time certain items in the copy of the header are initialized. Items not explicitly described as being initialized by the linker are set by the program that generates the object segment. In addition, the first two words of the header are filled in by the linker (when the header is copied) with a pointer to the beginning of the object segment's definition section. For more information see Section 4 and "Standard Stack and Linkage Area Formats" in Appendix H.

Linkage Section Header

The header of the linkage section (in an object segment) has the following format, defined in the system include file object_link_dcls.incl.pl1:

```
dcl 1 virgin_linkage_header       aligned based,
      2 pad                       bit(30) unal,
      2 defs_in_link              bit(6) unal,
      2 def_offset                fixed bin(18) uns unal,
      2 first_ref_relp            fixed bin(18) uns unal,
      2 filled_in_later           bit(144),
      2 link_begin                fixed bin(18) uns unal,
      2 linkage_section_lng       fixed bin(18) uns unal,
      2 segno_pad                 fixed bin(18) uns unal,
      2 static_length             fixed bin(18) uns unal;
```

STRUCTURE ELEMENTS

pad
is reserved for future use and must be 0.

defs_in_link
indicates whether or not there are definitions in the linkage section. If there are definitions in the linkage section, the value contained here is "010000"b.

def_offset
is an offset (relative to the base of the object segment) to the base of the definition section.

first_ref_relp
is an offset (relative to the base of the linkage section) to the first-reference trap. This trap is activated by the linker when the first reference to this object segment is made within a given process. If the value of this item is 0, there is no first-reference trap.

filled_in_later
is initialized by the linker when the header is copied. As a result of initialization by the linker, the first word becomes a pointer to the object segment's symbol section. It is used by the linker to snap links relative to the symbol section. The second word becomes a pointer to the original linkage section within the object segment. It is used by the link unsnapping mechanism. The last two words remain unused.

link_begin
is an offset (relative to the base of the linkage section) to the first link (the base of the link array).

linkage_section_lng
is the entire length in words of the entire linkage section.

segno_pad
is the segment number of the object segment. It is initialized by the linker when the header is copied.

static_length
is the length in words of the static section and is valid even when static is part of the linkage section. It is initialized by the linker if not filled in by the translator.

Internal Storage Area

The internal storage area is an array of words used by translators to allocate internal static variables and has no predetermined structure.

Links

A linkage section may contain an array of link pairs each of which defines an external name, referenced by this object segment, whose effective address is unknown at compile time. References to external entities are made by indirect references through a link, which has bee copied from the pure linkage section to the combined linkage section in the process directory.

Two types of links exist representing different levels of information about the target of a link.

Partially snapped links represent external references that have been resolved within an object MSF, but the target is not located in the same component segment as the reference. These links make use of information available to the binder about the target of the link so that they can be resolved by a first-reference trap procedure. Partially snapped links are distinguished from normal links in that they have a fault tag 3 modification initially. Once the first reference trap has run, the link has been replaced by an ITS pointer to the target of the link.

Normal links initially contain a fault tag 2 modification instead of an ITS modification. When the indirect reference is attempted,the fault occurs and is intercepted by the dynamic linking mechanism. Additional information in the link is used to locate the item referenced and, if successful, the link is replaced by an ITS pointer to the item. Figure G-3 illustrates the structure of a link.

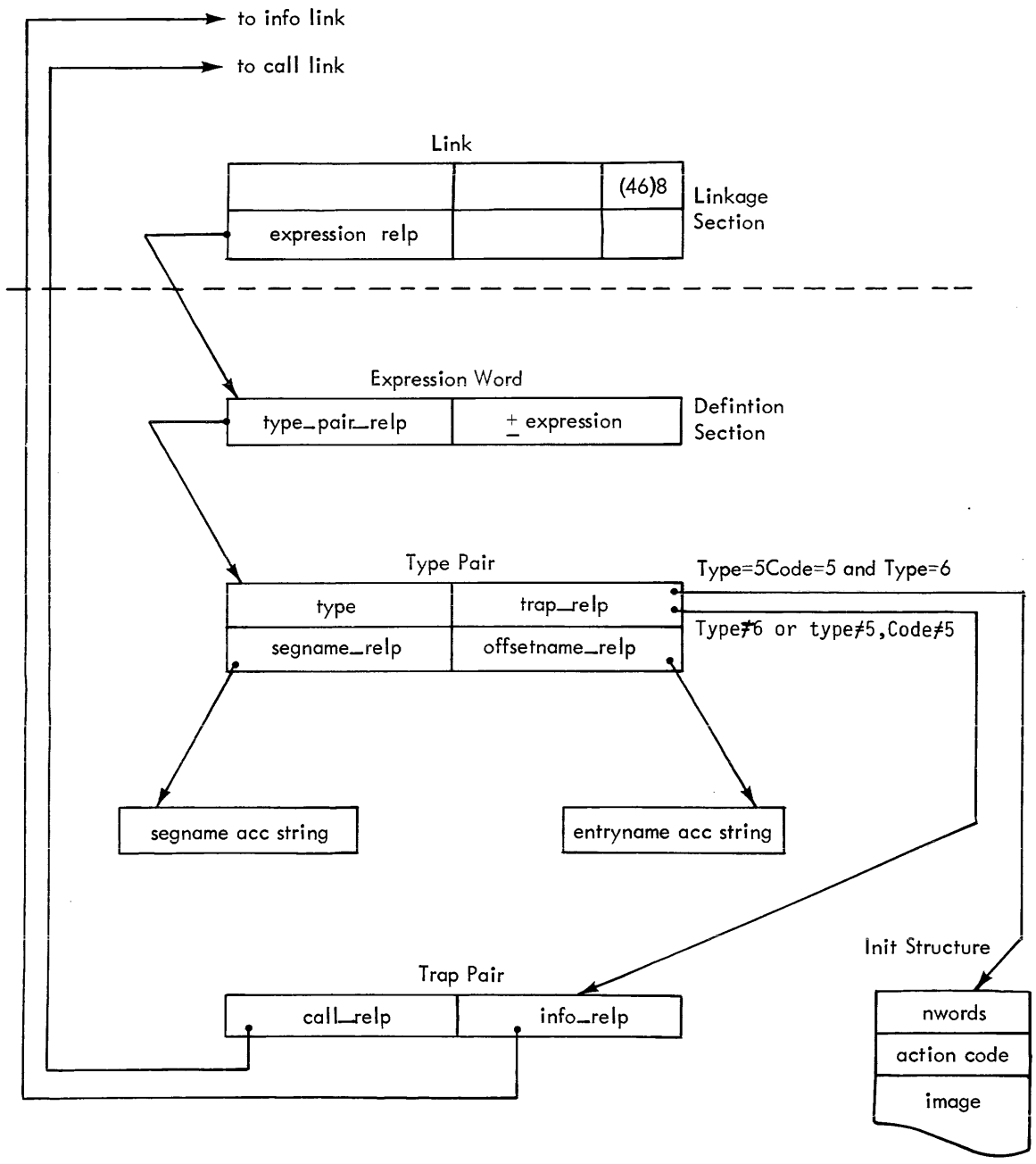


Figure G-3. Structure of a Link

A link must reside on an even location in memory, and must therefore be located at an even offset from the base of the linkage section. A link has the following format, defined in the system include file object_link_dcls.incl.pl1:

```
dcl 1 object_link          aligned based,
      2 header_relp        fixed bin(17) unal,
      2 ringno             fixed bin(3) uns unal,
      2 mbz                bit(3) unal,
      2 run_depth          fixed bin(5) unal,
      2 tag                bit(6) unal,
      2 expression_relp    fixed bin(18) uns unal,
      2 mbz2               bit(12) unal,
      2 modifier           bit(6) unal;
```

STRUCTURE ELEMENTS

header_relp
is an offset (relative to the link itself) to the head of the linkage section. It is, in other words, the negative value of the link pair's offset within the linkage section.

ringno
is the ring number of the ITS pointer.

mbz
is reserved for future use and must be "0"b.

run_depth
must be 0 in a generated (unsnapped) link. When the link is snapped, this field is filled in with the number of the current run unit level.

tag
is a constant (46)8 that represents the hardware fault tag 2 and distinctly identifies an unsnapped link. The snapped link (ITS pair) has a distinct (43)8 tag.

expression_relp
is an offset (relative to the base of the definition section) to the expression word for this link.

mbz2
is reserved for future use and must be "0"b.

modifier
is a hardware address modifier. When the link is snapped, this becomes the modifier of the ITS pair.

Partially Snapped Links

Partially snapped links are generated only within an object MSF to resolve a reference between components of the MSF. Partially snapped links are not snapped as a result of references to the link, but are all snapped at first reference to the MSF by a first reference trap placed in component 0. A partially snapped link must reside on an even location in memory, and must therefore be located at an even offset from the base of the linkage section. A partially snapped link has the following format, defined in the system include file object_link_dcls.incl.pl1:

```
dcl 1 partial_link          aligned based,
      2 type                fixed bin (3) unsigned unaligned,
      2 component           fixed bin (15) unsigned unaligned,
      2 mbz1                bit (12) unaligned,
      2 tag                 bit (6) unaligned,
      2 offset              fixed bin (18) unsigned unaligned,
      2 mbz2                bit (3) unaligned,
      2 bit_offset          fixed bin (6) unsigned unaligned,
      2 mbz3                bit (3) unaligned,
      2 modifier            bit (6) unaligned;
```

STRUCTURE ELEMENTS

type
This field is used to encode the target section of the link. The values used are the same as those used in a definition to indicate the target section.

- Text section = 0
- Linkage section = 1
- Symbol section = 2
- Static section = 4

component
is used to store the number of the target segment in the range 0 to component_count-1. At execution time, this value will be replaced depending on the target section as encoded in type.

```
Text section    - the component field will be replaced by
                  the segment number of the target segment.
Linkage section - the component field will be replaced by
                  the segment number of the segment containing
                  the copied linkage section.
Symbol section  - the component field will be replaced by the
                  segment number of the target segment.
Static section  - the component field will be replaced by the
                  segment number of the segment containing the
                  copied static section.
```

mbz1
Is unused and must be ""b.

tag
Is set to fault_tag_3 initially ("47"b3) to distinguish it from a normal unsnapped link or a snapped link with an ITS modifier ("43"b3). This is to insure that if the link does not get snapped somehow, a reference through it will fault. When the link is snapped, this field will be replaced by an ITS modifier.

offset
This field is the offset within the section identified by type to the target of the partial link. It is analogous to the thing_relp value in the corresponding definition in the target segment.

mbz2
Is unused and must be ""b.

bit_offset
Is the bit offset of the target of the link. Since a link target may not be on other than a word boundary, this field must be 0.

mbz3
Is unused and must be ""b.

modifier
Is the modifier of the link.

First-Reference Trap

It is sometimes necessary to perform certain types of initialization of an object segment when it is first referenced for execution (i.e., linked to) in a given process--for example, to store some per-process information in the segment before it is used. The first-reference trap mechanism provides this facility for use by various mechanisms, the status code assignment mechanism being an example.

A first-reference trap consists of two relative pointers. The first points to a link defining the first reference procedure entry point to be invoked. The second points to a link defining a block of information to be passed as an argument to the first-reference procedure. For more details on first-reference traps, see Section 4. The first reference trap has the following format, defined in the system include file object_link_dcls.incl.pl1:

```
dcl 1 fr_traps          aligned based,
      2 decl_vers       fixed bin,
      2 n_traps         fixed bin,
      2 trap_array      (0 refer(fr_traps.n_traps)) aligned,
        3 call_relp     fixed bin(18) uns unal,
        3 info_relp     fixed bin(18) uns unal;
```

STRUCTURE ELEMENTS

decl_vers
is the version number of the structure.

n_traps
specifies the number of traps.

trap_array
is an array of information about each first-reference procedure.

call_relp
is an offset (relative to the base of the linkage section) to a link defining a procedure to be invoked by the linker upon first reference to this object within a given process.

info_relp
is an offset (relative to the base of the linkage section) to a link specifying a block of information to be passed as an argument to the first reference procedure; if info_relp is 0, there is no such block.

STRUCTURE OF THE SYMBOL SECTION

The symbol section consists of one or more symbol headers threaded together to form a single list. A symbol header has two main functions: to document the circumstances under which the object segment was created, and to serve as a repository for information (relocation information, compiler's symbol tree, etc.) that does not belong in any of the other sections.

The symbol section must contain at least one symbol header, describing the circumstances under which the object segment was created. A symbol section can contain more than one symbol header. An example of multiple symbol headers is the case of a bound segment where in addition to the symbol header describing the segment's creation by the binder, there is also a symbol header for each of the component object segments.

Each symbol header can point to a free-format area. The free-format area can contain any information whatsoever, and the object segment will execute properly. However, the Multics debugging utilities (e.g., probe) place stringent requirements on the format of the free area, and these are followed by the translators for PL/I, FORTRAN, COBOL, and Pascal. See "Symbol Table Organization" below for additional information on the contents of the free-format area used by those three languages.

Symbol Block Header

All symbol blocks have a standard fixed-format block, although not all items in the block have meaning for all symbol blocks. The description of a particular symbol block lists items that have meaning for that symbol block. The block has the following format, defined by the system include file std_symbol_header.incl.pl1:

This page intentionally left blank.

```
dcl 1 std_symbol_header          based aligned,
    2 decl_version               fixed bin initial(1),
    2 identifier                 char(8) aligned,
    2 gen_number                 fixed bin,
    2 gen_created                fixed bin(71),
    2 object_created             fixed bin(71),
    2 generator                  char(8),
    2 gen_version                unaligned,
      3 offset                   bit(18),
      3 size                     bit(18),
    2 userid                     unaligned,
      3 offset                   bit(18),
      3 size                     bit(18),
    2 comment                    unaligned,
      3 offset                   bit(18),
      3 size                     bit(18),
    2 text_boundary              bit(18) unaligned,
    2 stat_boundary              bit(18) unaligned,
    2 source_map                 bit(18) unaligned,
    2 area_pointer               bit(18) unaligned,
    2 backpointer                bit(18) unaligned,
    2 block_size                 bit(18) unaligned,
    2 next_block                 bit(18) unaligned,
    2 rel_text                   bit(18) unaligned,
    2 rel_def                    bit(18) unaligned,
    2 rel_link                   bit(18) unaligned,
    2 rel_symbol                 bit(18) unaligned,
    2 mini_truncate              bit(18) unaligned,
    2 maxi_truncate              bit(18) unaligned;
```

STRUCTURE ELEMENTS

decl_version
: is the version number of the structure.

identifier
: is a symbolic name identifying the type of symbol block.

gen_number
: is a code designating the version of the generator that created this object segment. A generator's version number is normally changed when the generator or its output is significantly modified.

gen_created
: is a calendar clock reading specifying the date and time when this generator was created.

object_created
: is a calendar clock reading specifying the date and time when this symbol block was generated.

generator
is the name of the program that generated this symbol block.

offset
is an offset (relative to the base of the symbol block) to an aligned string describing the version of the generator. For example:

```
"PL/I Compiler Version 7.3
of Wednesday, July 28, 1971"
```

The integer part of the version number embedded in the string must be identical to the number stored in gen_number.

size
is the length of the aligned string describing the version of the generator.

userid
is the name of the user for whom this symbol block was created.

offset
is an offset (relative to the base of the symbol block) to an aligned string containing the access identification (i.e., the value returned by the get_group_id_ subroutine described in the Subroutines manual) of the user for whom this symbol block was created.

size
is the length of the aligned string containing the access identification of the user for whom the symbol block was created.

comment
an aligned string containing generator-dependent symbolic information. For example, a compiler might store diagnostic messages concerning nonfatal errors encountered while generating the object segment.

offset
is an offset (relative to the base of the symbol block) to the comment. A value of "0"b indicates no comment.

size
is the length of the aligned string containing generator-dependent symbolic information.

text_boundary
is a number indicating the boundary on which the text section must begin. For example, a value of 32 would indicate that the text section must begin on a 0 mod 32 word boundary. This value must be a multiple of 2. It is used by the binder to determine where to locate the text section of this object segment.

stat_boundary
is the same as text_boundary except that it applies to the internal static area of the linkage section of this object segment.

source_map
is an offset (relative to the base of the symbol block) to the source map (see "Source Map" below).

area_pointer
is an offset (relative to the base of the symbol block) to the free-format area of the symbol block. The contents of this area depend upon the symbol block. If the symbol block was created by a translator, this area may contain a runtime symbol table and/or a statement map. If the symbol block was created by the binder, this area contains the bind map.

backpointer
is an offset (relative to base of the symbol block) to the base of the symbol section; that is, the negative of the offset of the symbol block in the symbol section.

block_size
is the size of the symbol block (including the block) in words.

next_block
is a thread (relative to the base of the symbol section) to the next symbol block. This item is "0"b for the last block.

rel_text
is an offset (relative to the base of the symbol block) to text section relocation information (see "Relocation Information" below).

rel_def
is an offset (relative to the base of the symbol block) to definition section relocation information.

rel_link
is an offset (relative to the base of the symbol block) to linkage section relocation information.

rel_symbol
is an offset (relative to the base of the symbol block) to symbol section relocation information.

mini_truncate
is an offset (relative to the base of the symbol block) starting from which the binder systematically truncates control information (such as relocation bits) from the symbol section, while still maintaining such information as the symbol tree.

maxi_truncate
is an offset (relative to this base of the symbol block) starting from which the binder can optionally truncate nonessential parts of the symbol tree in order to achieve maximum reduction in the size of a bound object segment.

Source Map

The source map is a structure that uniquely identifies the source segments used to generate the object segment. It has the following format, defined in the system include file source_map.incl.pl1:

```
dcl 1 source_map                          aligned based,
    2 version                             fixed bin initial(1),
    2 number                              fixed bin,
    2 map (0 refer (source_map.number))   aligned,
      3 pathname                          unaligned,
        4 offset                          bit(18),
        4 size                            bit(18),
      3 uid                               bit(36) aligned,
      3 dtm                               fixed bin(71);
```

STRUCTURE ELEMENTS

version
is the version number of the structure.

number
is the number of entries in the map array; that is, the number of source segments used to generate this object segment.

pathname
an aligned string containing the absolute pathname of this source segment.

offset
is an offset (relative to the base of the symbol block) to the pathname.

size
is the length of the pathname.

uid
is the unique identifier of this source segment at the time the object segment was generated.

dtm
is the date-time-modified value of this source segment at the time the object segment was created.

Relocation Information

Relocation information, designating all instances of relative addressing within a given section of the object segment, enables the relocation of the section (as in the case of binding). A variable-length prefix coding scheme is used, where there is a logical relocation item for each halfword of a given section. If the halfword is an absolute value (nonrelocatable), that item is a single bit whose value is 0. Otherwise, the item is a string of either 5 or 15 bits whose first bit is set to "1"b. The relocation information is concatenated to form a single string that can only be accessed sequentially. If the next bit is a zero, it is a single-bit absolute relocation item; otherwise, it is either a 5- or a 15-bit item depending upon the relocation codes defined below.

There are four distinct blocks of relocation information, one for each of the four object segment sections: text, definition, linkage, and symbol; these relocation blocks are known as rel_text, rel_def, rel_link and rel_symbol, respectively.

The relocation blocks reside within the symbol block of the generator that produced the object segment. The correspondence between the packed relocation items and the halfwords in a given section is determined by matching the sequence of items with a sequence of halfwords, from left-to-right and from word-to-word by increasing value of address.

The relocation block pointed to from the symbol block block (e.g., text_relocation_relp) is structured as follows:

```
dcl 1 relinfo        aligned,
      2 decl_vers      fixed bin initial(2),
      2 n_bits         fixed bin,
      2 relbits        bit(0 refer(relinfo.n_bits)) aligned;
```

STRUCTURE ELEMENTS

decl_vers
is the version number of the structure.

n_bits
is the length (in bits) of the string of relocation bits.

relbits
is the string of relocation bits.

Following is a tabulation of the possible codes and their corresponding relocation types, followed by a description of each relocation type. Translators indicate the relocation code in the assembly-like listing of an object segment by a character. The second column below indicates the character used by standard translators. The third column indicates the character used by the ALM assembler. These codes may be found in relocation_bits.incl.pl1.

```
"0"b       - a  a  -   absolute
"10000"b   - t  0  -   text
"10001"b   - 1  1  -   negative text
"10010"b   - 2  2  -   link 18
"10011"b   - 3  3  -   negative link 18
"10100"b   - l  4  -   link 15
"10101"b   - d  5  -   definition
"10110"b   - s  6  -   symbol
"10111"b   - 7  7  -   negative symbol
"11000"b   - 8  8  -   internal storage 18
"11001"b   - i  9  -   internal storage 15
"11010"b   - r  L  -   self relative
"11011"b   -               unused
"11100"b   -               unused
"11101"b   -               unused
"11110"b   -               expanded absolute
"11111"b   - e  *          escape
```

absolute
: does not relocate.

text
: uses text section relocation counter.

negative text
: uses text section relocation counter. The reason for having distinct relocation codes for negative quantities is that special coding might be necessary to convert the 18-bit field in question into its correct fixed binary form.

link 18
: uses linkage section relocation counter on the entire 18-bit halfword. This, as well as the negative link 18 and the link 15 relocation codes apply only to the array of links in the linkage section (i.e., by definition, usage of these relocation codes implies external reference through a link).

negative link 18
: is the same as link 18 above.

link 15
: uses linkage section relocation counter on the low-order 15 bits of the halfword. This relocation code can only be used in conjunction with an instruction featuring a base/offset address field.

definition
: indicates that the halfword contains an address that is relative to the base of the definition section.

symbol
: uses symbol section relocation counter.

negative symbol
: is the same as symbol above.

internal storage 18
: uses internal storage relocation counter on the entire 18-bit halfword.

internal storage 15
: uses internal storage relocation counter on the low-order 15 bits of the halfword.

self relative
: indicates that the halfword contains a relocatable address that is referenced using a location counter modifier; the instruction is self-relocating.

expanded absolute
: allows the definition of a block of absolute relocated halfwords, for efficiency reasons. It has been established that a major part of an object program has the absolute relocation code. The five bits of relocation code are immediately followed by a fixed length 10-bit field that is a count of the number of contiguous halfwords all having an absolute relocation. Use of the expanded absolute code can be economically justified only if the number of contiguous absolute halfwords exceeds 15.

escape
reserved for possible future use.

STRUCTURE OF THE OBJECT MAP

The object map contains information used to locate the various sections of an object segment. The map itself can be located immediately before or immediately after any one of the five sections. Translators normally place it immediately after the symbol section. The last word of the object segment (as defined by the bit count of the object segment) must contain a left-justified 18-bit offset (relative to the base of the object segment) to the object map. The object map has the following format, defined in the system include file, object_map.incl.pl1:

```
dcl 1 object_map              aligned based,
      2 decl_vers             fixed bin init(2),
      2 identifier            char(8) aligned,
      2 text_offset           bit(18) unaligned,
      2 text_length           bit(18) unaligned,
      2 definition_offset     bit(18) unaligned,
      2 definition_length     bit(18) unaligned,
      2 linkage_offset        bit(18) unaligned,
      2 linkage_length        bit(18) unaligned,
      2 static_offset         bit(18) unaligned,
      2 static_length         bit(18) unaligned,
      2 symbol_offset         bit(18) unaligned,
      2 symbol_length         bit(18) unaligned,
      2 break_map_offset      bit(18) unaligned,
      2 break_map_length      bit(18) unaligned,
      2 entry_bound           bit(18) unaligned,
      2 text_link_offset      bit(18) unaligned,
      2 format                aligned,
        3 bound               bit(1) unaligned,
        3 relocatable         bit(1) unaligned,
        3 procedure           bit(1) unaligned,
        3 standard            bit(1) unaligned,
        3 separate_static     bit(1) unaligned,
        3 links_in_text       bit(1) unaligned,
        3 perprocess_static   bit(1) unaligned,
        3 unused              bit(29) unaligned;
```

STRUCTURE ELEMENTS

decl_vers
is the version number of the structure.

identifier
is the constant "obj_map".

text_offset
is an offset (relative to the base of the object segment) to the base of the text section.

text_length
is the length (in words) of the text section.

definition_offset
is an offset (relative to the base of the object segment) to the base of the definition section.

definition_length
is the length (in words) of the definition section.

linkage_offset
is an offset (relative to the base of the object segment) to the base of the linkage section.

linkage_length
is the length (in words) of the linkage section.

static_offset
is an offset (relative to the base of the object segment) to the base of the static section.

static_length
is the length (in words) of the static section.

symbol_offset
is an offset (relative to the base of the object segment) to the base of the symbol section.

symbol_length
is the length (in words) of the symbol section.

break_map_offset
is an offset (relative to the base of the object segment) to the base of the break _map section.

break_map_length
is the length (in words) of the break map section.

entry_bound
is the offset of the end of the entry transfer vector if the object segment is to be a gate.

text_link_relp
is the offset of the first text-embedded link if links_in_text equals "1"b.

bound
indicates if the object segment is a bound segment.

```
"1"b   the object segment is a bound segment
"0"b   the object segment is not a bound segment
```

relocatable
indicates if the object segment is relocatable; that is, if it contains relocation information. This information (if present) must be stored in the segment's first symbol block. See "Structure of the Symbol Section" above.

```
"1"b   the object segment is relocatable
"0"b   the object segment is not relocatable
```

procedure
indicates whether this is an executable object segment.

```
"1"b   this is an executable object segment
"0"b   this is not an executable object segment
```

standard
indicates whether the object segment is in standard format.

```
"1"b   the object segment is in standard format
"0"b   the object segment is not in standard format
```

separate_static
indicates whether the static section is separate from the linkage section.

```
"1"b   the static section is separate from the linkage section
"0"b   the static section is not separate from the linkage section
```

links_in_text
indicates whether the object segment contains text-embedded links.

```
"1"b   the object segment contains text-embedded links
"0"b   the object segment does not contain text-embedded links
```

perprocess_static
indicates whether the static section should be reinitialized for a run unit.

```
"1"b   static section is used as is
"0"b   static section is per run unit
```

unused
is reserved for future use and must be "0"b.

GENERATED CODE CONVENTIONS

The following discussion specifies those portions of generated code that must conform to a system-wide standard. For a description of the various relocation codes see "Structure of the Symbol Section" above.

Text Section

Those parts of the text section that must conform to a system-wide standard are:

```
entry sequence
text relocation codes.
```

ENTRY SEQUENCE

The entry sequence must fulfill two requirements:

1. The location preceding the entry point (i.e., entry point minus 1) must contain a left adjusted 18-bit relative pointer to the definition of that entry point within the definition section.

2. The entry sequence executed within that entry point must store an ITS pointer to that entry point in the entry_ptr field in the stack frame block (as described in the stack frame include file). The procedure's current stack frame can then be used to determine the address of the entry point at which it was invoked. That entry's symbolic name can be reconstructed through use of its definition pointer. (See "Entry Sequence" earlier in this section.)

TEXT RELOCATION CODES

The following list defines those relocation codes that can be generated in conjunction with the text section. These can be generated only within the scope of the restrictions specified.

absolute	no restriction
text	no restriction
negative text	no restriction
link 18	can only be a direct (i.e., unindexed) reference to a link.
link 15	can only appear within the address field of a pointer-register/offset type instruction (bit 29 = "1"b). The first two bits of the modifier field of the instruction cannot be "10"b. If the instruction uses indexing, the first two bits of the modifier must be "11"b. Also the following instruction codes cannot have this relocation code: STBA (551)8 STBQ (552)8 STCA (751)8 STCQ (752)8
definition	the offset to be relocated must be that of the beginning of a definition (relative to the beginning of the definition section).
symbol	no restriction
internal storage 18	no restriction

internal storage 15	can only apply to the left half of a word. If the word is an instruction, the first two bits of the modifier must not be "10"b.
self relative	no restriction
expanded absolute	no restriction

The restrictions imposed upon the link 15 and internal storage 15 relocation codes stem from the fact that these relocation codes apply to pointer-register/offset type address fields encountered in the address portion of machine instructions. Since the effective value of such an address is computed by the hardware at execution time, certain hardware restrictions are imposed on instructions containing them. When the Multics binder processes these instructions, it often resolves them into simple-address format and has to further modify information in the opcode (right-hand) portion of the instruction word. Therefore, these relocation codes must only be specified in a context that is comprehensible to the Multics processor.

Definition Section

Those parts of the definition section that must conform to a system-wide standard are:

```
general structure
definition relocation codes
implicit definitions
```

DEFINITION RELOCATION CODES

absolute	no restriction
text	no restriction
link 18	no restriction
definition	no restriction
symbol	no restriction
internal storage 18	no restriction
self relative	no restriction
expanded absolute	no restriction

IMPLICIT DEFINITIONS

All generated object segments must feature the following implicit definition:

symbol_table	defines the base of the symbol block generated by the current language processor, relative to the base of the symbol section.

Linkage Section

Those parts of the linkage section that must conform to a system-wide standard are:

```
internal storage
links
linkage relocation codes
```

INTERNAL STORAGE

The internal storage is a repository for items of the internal static storage class. It may contain data items only; it cannot contain any executable code.

LINKS

The link area can only contain a set of links. The links must be considered as distinct unrelated items, and no structure (e.g.. array) of links can be assumed. They must be accessed explicitly and individually through an unindexed internal reference featuring the link 18 or the link 15 relocation codes. The order of links will not necessarily be preserved by the binder.

LINKAGE RELOCATION CODES

Only the linkage section block and the links can have relocation codes associated with them (the internal storage area has associated with it a single expanded absolute relocation item). They are:

```
absolute              no restriction; mandatory for the
                        internal storage area
text                  no restriction
link 18               no restriction
negative link 18      no restriction
definition            no restriction
internal storage 18   no restriction
expanded absolute     no restriction
```

Static Section

The static section does not have relocation codes associated with it. Absolute relocation is assumed. See "Internal Storage Area" above.

Symbol Section

The symbol section can contain information related to some other section (such as a symbol tree defining addresses of symbolic items), and therefore can have relocation codes associated with it. They are:

```
absolute              no restriction
text                  no restriction
link 18               no restriction
definition            no restriction
symbol                no restriction
negative symbol       no restriction
internal storage 18   no restriction
self relative         no restriction
expanded absolute     no restriction
```

STRUCTURE OF BOUND SEGMENTS

A bound segment consists of several object segments that have been combined so that all internal intersegment references are automatically prelinked and to reduce the combined size by minimizing page breakage. The component segments are not simply concatenated; the binder breaks them apart and creates an object segment with single text, definition, static, linkage, and symbol sections as illustrated in Figure G-4 below. (When the static section is separate, it is located before the linkage block rather than between the linkage block and the links.) As explained below, the definition section and link array are completely reconstructed while the text, internal static, and symbol sections are the corresponding concatenations of the component segments' text, internal static, and symbol sections with relocation adjustments. (See "Structure of the Symbol Section" above.) If all of the components' static sections are separate (i.e., not in linkage), the bound segment has a separate static section; otherwise, all component static sections are placed in the bound segment's linkage section.

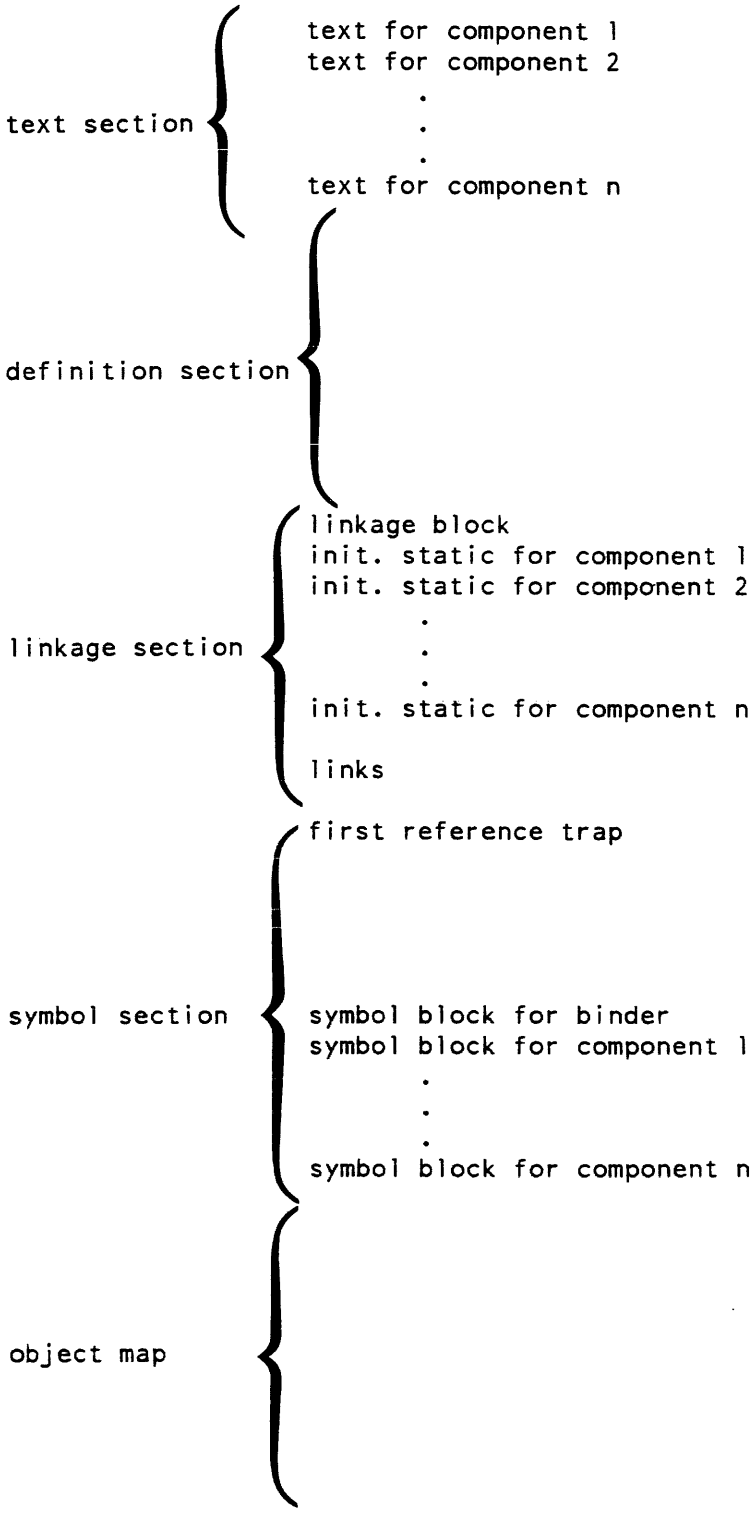


Figure G–4. Structure of a Bound Segment

Internal Link Resolution

The primary distinction between bound and unbound groups of segments occurs in the manner in which they reference external items and are themselves referenced. Most references by one component to another component in the same bound segment are prelinked; i.e., the link references are converted to direct text-to-text references and the associated links are not regenerated. The remaining external links are combined so that for the whole bound segment there is only one link for each different target. Prelinking enables some component segments to lose their identity in cases where the bound segment itself is the main logical entity, having been coded as separate segments for ease of coding and debugging. Definitions for external entries that are no longer necessary, i.e., have become completely internal, can be omitted from the bound segment (see the bind command described in the Commands manual).

Definition Section

The definition section of a bound segment is generally more elaborate than that of an unbound object segment because it reflects both the combination and deletion of definitions. There is a definition block for each component. It contains the retained definitions and the segment names associated with the component. This organization allows definitions for multiple entries with the same name to be distinguished. The first definition block is for the binder and contains a definition for bind_map, discussed below.

Binder Symbol Block

The symbol block of the binder has a standard block if all of the components are standard object segments. The symbol block can be located using the bind_map definition. Most of the items in the block are adequately explained under "Structure of the Symbol Section" above; however, some have special meaning for bound segments. The format of a standard symbol block block is repeated below for reference, followed by the explanations specific to the binder's symbol block.

```
dcl 1 std_symbol_block              based aligned,
      2 decl_version                fixed bin initial(1),
      2 identifier                  char(8) aligned,
      2 gen_number                  fixed bin,
      2 gen_created                 fixed bin(71),
      2 object_created              fixed bin(71),
      2 generator                   char(8),
      2 gen_version                 unaligned,
        3 offset                    bit(18),
        3 size                      bit(18),
      2 userid                      unaligned,
        3 offset                    bit(18),
        3 size                      bit(18),
      2 comment                     unaligned,
        3 offset                    bit(18),
        3 size                      bit(18),
      2 text_boundary               bit(18) unaligned,
      2 stat_boundary               bit(18) unaligned,
      2 source_map                  bit(18) unaligned,
      2 area_pointer                bit(18) unaligned,
      2 backpointer                 bit(18) unaligned,
      2 block_size                  bit(18) unaligned,
      2 next_block                  bit(18) unaligned,
      2 rel_text                    bit(18) unaligned,
      2 rel_def                     bit(18) unaligned,
      2 rel_link                    bit(18) unaligned,
      2 rel_symbol                  bit(18) unaligned,
      2 mini_truncate               bit(18) unaligned,
      2 maxi_truncate               bit(18) unaligned;
```

STRUCTURE ELEMENTS

identifier
is the string "bind_map".

generator
is the string "binder".

comment
is always "0"b.

area_pointer
is an offset (relative to the base of the symbol block) to the beginning of the bind map. (See "Bind Map" below.)

Bound segments currently are not relocatable, so none of the relocation relative pointers or truncation offsets have any meaning.

Bind Map

The bind map is part of the symbol block produced by the binder and describes the relocation values assigned to the various sections of the bound component object segments. It consists of a variable length structure followed by an area in which variable length symbolic information is stored. The bind map structure has the following format, defined in the system include file bind_map.incl.pl1:

```
dcl 1 bindmap              based aligned,
      2 dcl_version        fixed bin,
      2 n_components       fixed bin,
      2 component          (0 refer(bindmap.n_components)) aligned,
        3 name
          4 name_ptr       bit(18) unaligned,
          4 name_lng       bit(18) unaligned,
        3 comp_name        char(8) aligned,
        3 text_start       bit(18) unaligned,
        3 text_lng         bit(18) unaligned,
        3 stat_start       bit(18) unaligned,
        3 stat_lng         bit(18) unaligned,
        3 symb_start       bit(18) unaligned,
        3 symb_lng         bit(18) unaligned,
        3 defblock_ptr     unaligned,
        3 n_blocks         bit(18) unaligned,
      2 bf_name            aligned,
        3 bf_name_ptr      bit(18) unaligned,
        3 bf_name_lng      bit(18) unaligned,
      2 bf_date_up         char(24),
      2 bf_date_mod        char(24);
```

STRUCTURE ELEMENTS

dcl_version
is a constant designating the format of this structure; this constant is modified whenever the structure is, allowing system tools to easily differentiate bind map formats. This structure is version one (1).

n_components
is the number of component object segments bound within this bound segment.

component
is a variable-length array featuring one entry per bound component object segment.

name
is the symbolic name of the bound component. This is the name under which the component object was identified within the archive file used as the binder's input (i.e., the name corresponding to the object's objectname entry in the bindfile).

name_ptr
is the offset (relative to the base of the binder's symbol block).

name_lng
is the length (in characters) of the component's name.

comp_name
is the name of the translator that created this component object segment.

text_start
is the offset (relative to the base of the bound segment) of the component's text section.

text_lng
is the length (in words) of the component's text section.

stat_start
is the offset (relative to the base of the static section) of the component's internal static.

stat_lng
is the length of the component's internal static.

symb_start
is an offset (relative to the base of the symbol section) to the component's symbol section.

symb_lng
is the length of the component's symbol section.

defblock_ptr
if nonzero, this is a pointer (relative to the base of the definition section) to the component's definition block (first class-3 segname definition of that component's definition block).

n_blocks
is the number of symbol blocks in the component's symbol section.

bf_name_ptr
is the offset (relative to the base of the binder's symbol block) of the symbolic name of the bindfile.

bf_name_lng
is the length (in characters) of the bindfile name.

bf_date_up
is the date, in symbolic form, that the bindfile was updated in the archive (of object segments) used as input by the binder.

bf_date_mod
is the date, in symbolic form, that the bindfile was last modified before being put into the binder's object archive.

SYMBOL TABLE ORGANIZATION

The information below is subject to change. Future Multics releases may use a different format of runtime symbol information.

The free-format area can contain any information whatsoever, and the object segment will execute properly. However, the Multics debugging utilities (e.g., probe) place stringent requirements on the format of the free area, and these are followed by the translators for PL/I, FORTRAN, COBOL and Pascal.

The free-format area begins with a fixed-format header, called the pl1_symbol_block. Despite the name, this block is present even in FORTRAN, Pascal and COBOL-produced object segments. The pl1_symbol_block gives the options used in compiling the segment, and the offsets of the statement map, the root block node, and the profile information.

The remainder of the free-format area consists of the statement map, the symbol tree, and the profile information, which are discussed below.

The PL/I Symbol Block

The PL/I symbol block has the following format (declared in pl1_symbol_block.incl.pl1):

```
dcl 1 pl1_symbol_block   aligned,
    2 version            fixed binary,
    2 identifier         char(8),
    2 flags,
      3 profile          bit(1) unaligned,
      3 table            bit(1) unaligned,
      3 map              bit(1) unaligned,
      3 flow             bit(1) unaligned,
      3 io               bit(1) unaligned,
      3 table_removed    bit(1) unaligned,
      3 long_profile     bit(1) unaligned,
      3 pad              bit(29) unaligned,
    2 greatest_severity  fixed binary,
    2 root               bit(18) unaligned,
    2 profile            bit(18) unaligned,
    2 map,
      3 first            bit(18) unaligned,
      3 last             bit(18) unaligned,
    2 segname,
      3 offset           bit(18) unaligned,
      3 length           bit(18) unaligned;
```

STRUCTURE ELEMENTS

version
is the version number of the structure. For this version the version number is 1.

identifier
is the constant "pl1info".

profile
is "1"b if the object program contains an execution profile table. This table is generated if the -profile control argument is specified when the source program is compiled.

table
is "1"b if the object program contains a runtime symbol table. A runtime symbol table is generated if the -table control argument is specified when the source program is compiled or if the runtime table is required by PL/I put data or get data or FORTRAN namelist input/output statements in the source program (see "The PL/I Runtime Symbol Table" below).

map
is "1"b if the object segment contains a statement map that gives the correspondence between source line numbers and locations in the object segment (see "The Statement Map" below). The statement map is present if the -brief_table, -profile, or -table control arguments are specified when the source program is compiled.

flow
is "1"b if the object program contains additional instructions for monitoring program flow. This facility is not yet available.

io
is "1"b if the object program contains a runtime symbol table that is required by PL/I put data or get data or FORTRAN namelist input/output statements in the source program. In this case the runtime symbol table cannot be removed.

table_removed
is "1"b if the object segment originally contained a runtime symbol table that has subsequently been removed.

long_profile
is "1"b if the object segment contains a long profile table.

greatest_severity
contains the greatest severity level of all error messages issued during the compilation of the source program. A value of 0 means that no errors were found during compilation.

root
is nonzero only if the object segment contains a runtime symbol table; in this case, root is a pointer (relative to the base of the symbol header block) to the root block of the runtime symbol table.

profile
is nonzero if the object segment contains a profile table. If it is nonzero, it is the offset in the linkage section of the table.

first
is nonzero only if the object segment contains a statement map; in this case, first is a pointer (relative to the base of the symbol header block) to the first entry in the statement map.

last
is nonzero only if the object segment contains a statement map; in this case, last is a pointer (relative to the base of the symbol header block) to the last entry in the statement map.

offset
is a pointer (relative to the base of the symbol header block) to an aligned character string that gives the name of the segment; this is the same as the name used for the class 3 definition of the object segment.

size
is the length of the segment name string.

The PL/I Runtime Symbol Table

The PL/I runtime symbol table contains information needed to support source language debugging and PL/I data-directed or FORTRAN namelist input/output statements. Most of the information that the compiler has in its compile-time symbol table is placed, in a different format, in the runtime symbol table; this permits attributes of a variable such as data type, storage class, or location to be determined during execution of the program. If the runtime symbol table is present, it follows the PL/I symbol block.

There are two types of runtime symbol tables: partial tables and full tables.

A partial table is generated when the source program contains data-directed input/output statements; it contains information only about variables that are transmitted via PL/I data-directed or FORTRAN namelist input/output statements. A partial runtime symbol table cannot be removed.

A full symbol table is generated if the table control argument is specified when the source program is compiled; it contains information about all variables, labels, and entries referenced by the source program. A full symbol table can be removed from the object program (when binding) if the source program does not contain data-directed input/output statements that would require a partial table to be generated.

The existence of a runtime symbol table does not affect the executable code normally generated by the compiler. There are no instructions that must routinely be executed by the object program in order to support the runtime symbol table. In some cases (described later), the compiler generates additional code sequences solely because a runtime symbol table is being created, but these extra instructions are not executed unless particular fields of the runtime symbol table are actually referenced.

An internal static variable that has an initial value and is never set is normally treated just as if it were a constant. If all references to the value of the internal static variable can be made using DU or DL modifiers in the instructions making the reference, the variable is not assigned a location. If all references cannot be made via DU or DL modifiers, the variable is assigned one or more locations in the text section. When a runtime symbol table is being generated, internal static variables that are initialized and never set are always assigned locations in the text section. This does not affect references to these variables since DU or DL modifiers continue to be used wherever possible.

The runtime symbol table is a list structure that consists of interconnected runtime_token, runtime_block, and runtime_symbol nodes. Normally, when node A in the runtime symbol table contains a pointer to node B, the pointer is relative to the start of the node in which it occurs; such a pointer is called a *self-relative pointer.* The format of the nodes in the runtime symbol table are described in the sections that follow.

THE RUNTIME_TOKEN NODE

The runtime_token node holds the name of an identifier used elsewhere in the runtime symbol table. The runtime_token nodes for all identifiers in the runtime symbol table are threaded together on a list that is ordered alphabetically by size (all 1 character names before all 2 character names, etc.); there are no duplicate names on this list. This ordering is used to increase the speed with which the runtime symbol table can be searched. Each runtime_token node contains a pointer to the runtime_symbol node for the first variable having the name stored in the runtime_token node. The runtime_token node has the following format (and appears in runtime_symbol.incl.pl1):

```
dcl 1 runtime_token    based aligned,
      2 next           bit(18) unaligned,
      2 dcl            bit(18) unaligned,
      2 name,
        3 size         fixed bin(9) unsigned unaligned,
        3 string       char(0 refer(runtime_token.size))
                                    unaligned;
```

STRUCTURE ELEMENTS

next
: is a self-relative pointer to the next token on the alphabetic by size list of tokens. This field is zero in the last runtime_token node on the list.

dcl
: is a self-relative pointer to the runtime_symbol node for the first identifier having the name stored in this runtime_token node. This field is zero if there are no identifiers declared with this name.

name
: is an ACC string that gives the name of the identifier represented by this node (see "The Structure of the Definition Section" above for a description of ACC strings).

THE RUNTIME_BLOCK NODE

Each procedure or begin block in the source program has a corresponding runtime_block node in the runtime symbol table. The manner in which these nodes are connected reflects the block structure of the source program. Each runtime_block node contains a pointer to a list of runtime_symbol nodes that represent declarations defined immediately internal to the block (i.e. internal to the block but not internal to any other block contained in the block). These declarations correspond to the variables and label or entry constants used in the block. The runtime_block node has the following format (which appears in runtime_symbol.incl.pl1):

```
dcl 1 runtime_block      aligned,
      2 flag             bit(1) unaligned,
      2 quick            bit(1) unaligned,
      2 fortran          bit(1) unaligned,
      2 standard         bit(1) unaligned,
      2 owner_flag       bit(1) unaligned,
      2 skip             bit(1) unaligned,
      2 type             bit(6) unaligned,
      2 number           bit(6) unaligned,
      2 start            bit(18) unaligned,
      2 name             bit(18) unaligned,
      2 brother          bit(18) unaligned,
      2 father           bit(18) unaligned,
      2 son              bit(18) unaligned,
      2 map,
        3 first          bit(18) unaligned,
        3 last           bit(18) unaligned,
      2 entry_info       bit(18) unaligned,
      2 header           bit(18) unaligned,
      2 chain(4)         bit(18) unaligned,
      2 token(0:5)       bit(18) unaligned,
      2 owner            bit(18) unaligned;
```

STRUCTURE ELEMENTS

flag
: is always "1"b and is used to tell this version of the structure from an earlier one.

quick
: is "1"b if the procedure or begin block that corresponds to this runtime_block node is a quick block that does not have a stack frame of its own. By definition, when a quick block is called, pr6 (the stack pointer) points at the stack frame shared by the quick block in which the quick block allocates its storage. This bit is always "0"b in the runtime_block that corresponds to an external procedure.

fortran
is "1"b if this program was compiled by the FORTRAN compiler. This bit is used to tell the programs that access the runtime symbol table that array elements are stored in column-major order instead of row-major order. The object program contains other places that indicate the compiler that processed the program; this bit was added to increase the speed with which this information could be obtained.

standard
is "1"b if this object segment is in standard Multics format. Here, too, information that is available elsewhere is repeated for the sake of convenience.

owner_flag
is "1"b if this block has a valid owner field.

skip
is reserved for future expansion.

type
indicates what kind of block the block node corresponds to. The following values are defined:
"01"b3 external entry
"02"b3 nonquick internal procedure
"03"b3 quick internal procedure
"04"b3 begin block
"05"b3 Pascal with block

(Currently this field is not always filled in as defined above.)

number
is used to number begin blocks. All begin blocks in the source program are assigned a sequence number in the order in which they are encountered by the program that generates the runtime symbol table.

start
is a self-relative pointer to the runtime_symbol node for the first declaration in the block represented by the runtime_block node. This declaration list gives all level 0 (nonstructure) and level 1 (top level structure) symbols defined immediately internal to the block; the runtime_symbol nodes on this list are ordered alphabetically by size. The start field is zero if there are no declarations in the block.

name
is a self-relative pointer to the ACC string that gives the name of the block; this field is zero for a begin block. The block compiled for an on-unit is a procedure block whose name is derived from the name of the condition, e.g. "overflow.1". For historical reasons, the name component points at runtime_token.name instead of the beginning of runtime_token.

brother
is a self-relative pointer to the next runtime_block node at the same nesting level. This field is zero if there is no other block at the same nesting level.

father
is a self-relative pointer to the immediately containing runtime_block node of which this block is a son. If the current block is the root of the symbol tree, this pointer points to the symbol header block.

son
is a self-relative pointer to the first runtime_block node contained within the current block. This field is zero if the current block does not contain any other blocks.

first
is nonzero if the object program contains a statement map; in this case first is a self-relative pointer to the entry in the statement map that corresponds to the first executable statement in this block. If block B is contained in block A, the entries in the statement map for block B are also contained in the statement map entries for block A.

last
is a self-relative pointer to the word after the entry that corresponds to the last executable statement. Note that zero is a meaningful value.

entry_info
is nonzero only for a runtime_block that corresponds to a procedure without its own stack frame (quick = "1"b). It gives the location in the stack frame shared by the quick block of the entry information block used by the quick block. The format of an entry information block is described below.

header
is a self-relative pointer to the start of the symbol header block.

chain
is a vector of self-relative pointers that point at runtime_symbol nodes on the declaration list for this block. The chain(i) points at the runtime_symbol node for the first declaration whose name is longer than 2**i; chain(i) is zero if the longest name in the declaration list is shorter than 2**i.

token
is a vector of self-relative pointers that point at runtime_token nodes. The token(i) points at the runtime_token node for the first name longer than 2**i; token(i) is zero if the longest name in the token list is shorter than 2**i.

owner
is a self-relative pointer to the runtime_block node whose stack frame will be shared by this block. This field is valid only if owner_flag is set.

THE ENTRY INFO BLOCK

An entry info block consists of one, two, or three pointers, depending on the procedure. It has the following format (declared in quick_entry.incl.pl1):

```
dcl 1 quick_entry        aligned,
      2 return           ptr,
      2 argptr           ptr,
      2 descptr          ptr;
```

STRUCTURE ELEMENTS

return
points at the return location of the quick block.

argptr
if present, points at the argument list of the quick block.

descptr
if present, points at the descriptor list of the quick procedure.

THE PASCAL "with" BLOCK

When the block node corresponds to a Pascal "with" block (runtime_block = "05"b3), it has two more fields appended to it. The node then has the following format (which appears in runtime_symbol_block.incl.pl1, where symbol_block is equivalent to the runtime_block structure described above):

```
dcl 1 with_symbol_block  aligned,
      2 common_block_info
                         aligned like symbol_block,
      2 with_string      fixed bin (18) unsigned unaligned,
      2 real_level_1     fixed bin (18) unsigned unaligned;
```

STRUCTURE ELEMENTS

with_string
is a self-relative pointer to the ACC string that duplicates the string in the source program's "with" statement.

real_level_1
is a self-relative pointer to the level 1 node for the corresponding record type.

THE RUNTIME_SYMBOL NODE

Each runtime_symbol node in the runtime symbol table corresponds to an identifier in the source program. The manner in which these nodes are connected reflects the structural relationship of variables in the source program. Level 0 (nonstructure) and level 1 (top level structure) variables have the runtime_symbol nodes that correspond to them threaded on a list of runtime_symbol nodes ordered alphabetically by size.

The format of the runtime_symbol node is (declared in runtime_symbol.incl.pl1):

```
dcl 1 runtime_symbol      aligned,
      2 flag              bit(1) unaligned,
      2 use_digit         bit(1) unaligned,
      2 array_units       bit(2) unaligned,
      2 units             bit(2) unaligned,
      2 type              bit(6) unaligned,
      2 level             bit(6) unaligned,
      2 ndims             bit(6) unaligned,
      2 bits              unaligned,
        3 aligned         bit(1),
        3 packed          bit(1),
        3 simple          bit(1),
        3 decimal         bit(1),
       2 scale            bit(8) unaligned,
       2 name             bit(18) unaligned,
       2 brother          bit(18) unaligned,
       2 father           bit(18) unaligned,
       2 son              bit(18) unaligned,
       2 address          unaligned,
         3 location       bit(18),
         3 class          bit(4),
         3 next           bit(14),
       2 size             fixed binary(35),
       2 offset           fixed binary(35),
       2 virtual_org      fixed binary(35),
       2 bounds(1),
         3 lower          fixed binary(35),
         3 upper          fixed binary(35),
         3 multiplier     fixed binary(35);
```

In the discussion that follows, the term "current identifier" means the indentifier represented by the runtime_symbol node under consideration, and the term "current block" means the block in which the current identifier is declared:

STRUCTURE ELEMENTS

flag
is always "1"b and distinguishes this version of the structure from other versions.

use_digit
contains the most significant bit of the three bit binary integers that identify the addressing units for arrays and offsets.

array_units
contains the low order two bits of a three bit positive binary integer that gives the addressing units to be used when computing the address of a subscripted array element; this field is meaningful only when ndims is not zero. The high order bit is supplied by the use_digit bit. The possible values for this three bit number, and the corresponding factor by which an offset should be multiplied to convert to a bit offset are:

units	*factor*
0 word	36
1 bit	1
2 byte	9
3 half word	18
4 word	36
5 bit	1
6 byte	9
7 digit	4.5

units
contains the low order two bits of a positive binary integer that gives the addressing units of the offset field in the runtime_symbol node. The high order bit is supplied by use_digit. The possible values and associated conversion factors are the same as for array_units.

type
contains a positive binary integer that gives the data type of the current identifier. The numeric values used to encode the data type are the same as the values used in the Multics descriptor, supplemented with additional values (see Appendix D).

When the identifier is a pictured variable, the real data type is given by the picture information block, which can be found by using information in the size field of the runtime_symbol node.

level
contains a positive binary integer that gives the structure nesting level of the current identifier as determined by the compiler; nonstructure variables have level = 0.

ndims
contains a positive binary integer that gives the number of array dimensions of the current identifier; a value of zero means the current identifier is not an array. The ndims gives the total number of subscripts that must be provided to access an element of the array and is the sum of the number of dimensions with which the identifier was explicitly declared and the number of dimensions inherited from a containing structure.

aligned
is "1"b if the current identifier is aligned and is "0"b if the identifier is unaligned.

packed
is "1"b if the current identifier is any one of the following: an unaligned aggregate of packed data, unaligned arithmetic data, unaligned nonvarying string data, or unaligned pointer data.

simple
is "1"b if an abbreviated form of the runtime_symbol node is being used for the current identifier; in this case fields after size in the runtime_symbol node are *not* present and the current identifier is a scalar with zero offset. If simple is "0"b, *all* fields in the runtime_symbol node are present.

decimal
is reserved for future expansion.

scale
is the arithmetic scale factor of the current identifier. Although stored in a bit (8), it is logically a fixed bin (7). Be warned that COBOL and PL/I both define negative scale factors, and that PL/I bit to fixed conversion assumes unsigned, not signed.

name
is a self-relative pointer to the ACC string that gives the name of the current identifier. For historical reasons, the name component points at runtime_token.name instead of the beginning of runtime_token.

brother
is a self-relative pointer to the runtime_symbol node for the next identifier at the same structure level; levels 0 and 1 are considered to be the same level. Within a structure (level > 1), brother points to the runtime_symbol node for the identifier that immediately follows the current identifier in the structure; brother is zero if the current identifier is the last element in the structure that immediately contains it. Outside of a structure (level <= 1), brother points to the next element on the list of runtime_symbol nodes ordered alphabetically by size.

father
is a self-relative pointer to either a runtime_block node or a runtime_symbol node. If level <= 1, father points to the runtime_block node that represents the block in which the current identifier is declared. If level > 1, father points to the runtime_symbol node for the structure that immediately contains the current identifier as a son.

son
is a self-relative pointer to the first son of a structure (the runtime_symbol node for the first identifier in the structure with a level number one greater than the level of the current identifier). This field is zero if the current identifier is not a structure.

location
usually contains a positive integer L that is used in combination with class to determine the address of the current identifier. L is normally an offset with respect to the start of a given class of storage; its interpretation depends on the value of the class field in the runtime_symbol node.

class
contains a positive binary integer that gives the storage class of the current identifier; the possible classes are:

class	*storage class*
1	automatic; L is the offset at which the current identifier is defined in the stack frame associated with the current block.
2	automatic adjustable; the address of the current identifier is not known at the time the runtime symbol table is created. Location L in the stack frame associated with the current block contains a pointer to the storage for the current identifier.
3	based; location is a self-relative pointer to the runtime_symbol for the pointer used in the declaration of the current identifier or is zero if a pointer was not specified. The user must provide a pointer, either explicitly at run time or implicitly through the default pointer, in order to reference the current identifier.
4	internal static; L is the offset at which the current identifier is assigned storage in the linkage section associated with the current block.
5	external static; L is the offset in the linkage section of a link that points to the current identifier.
6	internal controlled; L is the offset of the control block of the current identifier in the linkage section of the current block.
7	external controlled; L is the offset in the linkage section of a link that points to the control block for the current identifier.
8	parameter; at L in the stack frame corresponding to the current block there is a pointer to the storage for the current identifier. This storage class is used when the current identifier appears in more than one position in procedure and/or entry statements in the block.

9 parameter; L gives the position of the current identifier in the argument list provided to the current block. This class is used when the current identifier appears in the same position in every procedure or entry statement in the current block.

10 very large array; L is a self-relative pointer to the runtime_symbol for the pointer to the beginning of the array. Address arithmetic must be used to calculate offsets from this base.

11 symbol table constant; location is a self-relative pointer to the value.

12 text reference; the current identifier is defined at L in the text section of the object segment.

13 link reference; the current identifier is defined at L in the linkage section corresponding to the current block.

14 not used

15 not used

next
is a self-relative pointer to the runtime_symbol node of the next identifier having the same name as the current identifier.

size
is the arithmetic precision, string size, or area size of the identifier. If the identifier is a string or area, it may be an encoded value. If the current identifier is a picture variable, size contains the offset at which the picture information block can be found in the text section of the object segment. If the current identifier is an offset variable, size is a self-relative pointer to the runtime_symbol node for the area, if any, associated with the current identifier.

offset
is the encoded value of the offset of the start of the current identifier with respect to the address specified by location and class. The units of the offset value are given by the units field in the runtime_symbol node. This field is not present, and its value is assumed to be zero, if the simple bit is "1"b.

virtual_org
is the encoded value of the virtual origin of an array, in units given by array_units. Its value should be subtracted from the base address specified by location and class. This field is not present, and the current identifier is a scalar, if the simple bit is "1"b.

bounds
is an array that gives information about each dimension of an array identifier, from left to right. The upper bound for the bounds array that appears in the declaration is actually a dummy; the true upper bound for the bounds array is given by the ndims field. All the fields in the bounds array are not present, and the current identifier is a scalar, if the simple bit is "1"b. A bound structure is declared in runtime_bound in runtime_symbol.incl.pl1.

lower
is the encoded value of the lower bound of this dimension of the current identifier.

upper
in the encoded value of the upper bound of this dimension of the current identifier.

multiplier
is the encoded value of the multiplier of this dimension of the current identifier.

The address of an identifier is calculated in the following manner. The base address is determined by the class and location fields. If the identifier is "simple", this is all. Otherwise, the offset field (which may be encoded) is multiplied by the conversion factor given by use_digit and units to give a bit offset, which is added to the base address. If the identifier is not an array element, that is all; otherwise, the virtual origin is computed (an encoded value converted to bits by the factor given by use_digit and array_units) and subtracted from the address. The array offset is computed by taking the dot product of the subscripts supplied and the multipliers for the identifier. The array offset is converted to a bit offset using the array_units conversion factor, and added to the address previously computed. This gives the final address of the data.

Encoded Values

The runtime_symbol node contains information about the attributes of an identifier. In many cases, the value of attributes such as string length, array bounds, or address cannot be determined at the time the runtime symbol table is created. For example, given the declaration:

```
dcl x char (n+m) ;
```

the length of the variable x can be different each time the block in which it is declared is entered; the location of x is not known because a variable with nonconstant size is allocated when the block is entered. If x were declared instead:

```
dcl x char (n+m) based;
```

the length of x could be different at each reference.

The problem of representing nonconstant attributes values is handled by encoding the values that can be nonconstant. A field in the runtime_symbol node that can have a nonconstant value is called an *encoded value*; it is declared fixed binary(35) in the node declaration, but actually has the following format (declared in runtime_symbol.incl.pl1):

```
dcl 1 encoded_value      aligned,
      2 flag             bit(2)  unaligned,
      2 code             bit(4)  unaligned,
      2 (n1,n2)          bit(6)  unaligned,
      2 n3               bit(18) unaligned;
```

If flag ^= "10"b, the encoded value is the constant given in the entire word. If flag = "10"b, the positive binary integer contained in the code field determines the value.

In Pascal symbol nodes, it has the following format (declared in pascal_symbol_node.incl.pl1):

```
dcl 1 pascal_encoded_value aligned,
      2 code                bit(6) unaligned,
      2 (n1,n2)             bit(6) unaligned,
      2 n3                  fixed bin (18) unsigned unaligned;
```

In the Pascal symbol node declaration, values that may be encoded are also declared as fixed binary (35). However in this case, the flags that indicate encoding are outside the encoded value.

The code values are defined as follows:

Code	*Value*
0	Value is the contents of the word at location n3 in the stack frame of the block n1 static levels before the block in which the declaration occurs.
1	Value is the contents of the word at location n3 in the linkage section of the block in which the declaration occurs.
2	Value is the contents of the word with positive offset n1 from the word pointed at by the link at location n3 in the linkage section of the block in which the declaration occurs.
3	Value is n3 plus the contents of the bit offset field of the pointer used to access the variable, which must be based. This encoding was only used by the compiler before version 2 EIS.
4	Value is the contents of the word with positive offset n2 based on the pointer at location n3 in the stack frame n1 static levels before the block in which the declaration occurs.
5	Value is the contents of the word with positive offset n2 based on the pointer at location n3 in the linkage section of the block in which the declaration occurs.
6	Value is the contents of the word with positive offset n2 based on the pointer with positive offset n1 from the word pointed at by the link at location n3 in the linkage section of the block in which the declaration occurs.
7	Value is the contents of the word with positive offset n2 based on the pointer used to access the variable, which must be based. This encoding is used for refer extents.

8 Value is the value returned by the internal procedure at location n3 in the text section of the block in which the declaration occurs. This procedure is compiled as if it were declared in the block in which the declaration occurs. This encoding is used whenever one of the other more specific encodings cannot be used. The calling sequence of this procedure is:

```
dcl f entry(ptr) returns(fixed binary(24));
value = f(refp);
```

where refp is the pointer that could be used to access a based variable. Note that this procedure is never called by the executable code in the object program, it is used only by the programs that reference the runtime symbol table.

9 Value is the contents of the word with positive offset n3 from the start of argument n2 of the procedure n1 static levels before the block in which the declaration occurs.

10 Value is the contents of the word with positive offset n3 from the word pointed at by the pointer that is argument n2 of the procedure n1 static levels above the block in which the declaration occurs.

11 Value is the contents of the size field of descriptor n2 of the procedure n1 static levels before the block in which the declaration occurs.

12 Value is the contents of the word with positive offset n3 from the start of descriptor n2 of the procedure n1 static levels before the block in which the declaration occurs.

13 Value is the size field at positive offset n2 from the start of the descriptor for a controlled variable. For all encodings having to do with controlled variables, if n1 = 0 the variable is internal, if n1 = 1 it is external. For an internal controlled variable a pointer to the descriptor (control_block.descriptor) is located at n3 in the static secion. For an external variable, a ptr to the descriptor ptr is at n3 in the linkage section.

14 Value is the contents of the word with positive offset n2 from the start of the descriptor for a controlled variable. The descriptor is located in the same manner used for type 13 encoding.

15 Value is the contents of the word with positive offset n2 from the start of a controlled variable. If n1 = 0 the controlled variable is internal and its control block is located at n3 in the linkage section of the block in which the declaration occurs. If n1 = 1 the controlled variable is external and location n3 in the linkage section of the block in which the declaration occurs contains a pointer to the control block. The data itself is found using the data pointer of the controlled variable control block.

16 Value is the contents of the location described by the symbol node pointed to by self-relative pointer n3. n1 = "01"b3 if the value is signed; n1 = ""00"b3 if the value is unsigned. n2 is the precision in bits of the value.

Controlled Variable Control Block

The format of the control block for a controlled variable is given in ctl_block.incl.pl1:

```
dcl  1 control_block     aligned,
       2 data  ptr,
       2 descriptor      ptr,
       2 previous        ptr;
```

Structure Elements

data
points at the current generation of the controlled variable. It is null if the controlled variable does not have a current generation.

descriptor
points at the descriptor for the current generation of the controlled variable.

previous
points at the control block of the previous generation of the controlled variable. It is null or points to a null ptr if there is no previous generation.

Picture Information Block

A picture variable of any type is stored in edited form as a character string. Each picture variable has an "associated value" that gives the value of the picture variable in internal form, either as a character string or as a decimal number. When the current identifier is a picture variable, the size field in the runtime_symbol node specifies the location of the picture information block, whose format is (declared in picture_image.incl.pl1):

```
dcl 1 picture_info    based aligned,
      2 type          fixed binary(8) unaligned,
      2 prec          fixed binary(8) unaligned,
      2 scale         fixed binary(8) unaligned,
      2 piclength     fixed binary(8) unaligned,
      2 varlength     fixed binary(8) unaligned,
      2 scalefactor   fixed binary(8) unaligned,
      2 explength     fixed binary(8) unaligned,
      2 drift         char(1) unaligned,
      2 chars         char(0 refer(picture_info.piclength)) aligned;
```

type
is the true data type of the current identifier according to the following encoding:

type	*data type*	*named constants in picture_image.incl.pl1*
24	character string	picture_char_type
25	real fixed decimal	picture_realfix_type
26	complex fixed decimal	picture_complexfit_type
27	real float decimal	picture_realflo_type
28	complex float decimal	picture_complexflo_type

prec
is the arithmetic precision or string length of the associated value. Note that the length of a character picture variable must be constant.

scale
for arithmetic picture variables is the number of digits, if any, after the "v" in the picture constant minus scale factor (see below).

piclength
is the length of the normalized picture constant string.

varlength
is the length of the edited form of the picture variable in characters. Note that the length of a picture variable must be constant.

scalefactor
is the picture scale factor.

explength
is the length in characters of the exponent field of a floating point picture variable.

drift
is the picture drifting character. It is blank if the picture constant does not specify a drifting field.

chars
is the normalized picture constant.

The Pascal Runtime Symbol Node

Pascal runtime symbol nodes are similar in function to PL/I runtime symbol nodes. However, some of these nodes correspond to types rather than variables or constants. The format of the Pascal runtime symbol node is shown below as a series of separate structure declarations. No single node will contain all the items. The header is always present. Bit flags in the header indicate which of the additional items are present. The additional items always follow contiguously in the order shown. (The declarations below appear in pascal_symbol_node.incl.pl1.)

```
dcl 1 pascal_symbol_node_header aligned based,
      2 flags                  unal,
        3 version_flag         bit (1) unal,
        3 aligned              bit (1) unal,
        3 packed               bit (1) unal,
        3 in_with_block        bit (1) unal,
        3 name_next            bit (1) unal,
        3 base_type_info       bit (1) unal,
        3 address              bit (1) unal,
        3 father_brother       bit (1) unal,
        3 son_level            bit (1) unal,
        3 father_type_successor bit (1) unal,
        3 size                 bit (1) unal,
        3 offset               bit (1) unal,
        3 subrange_limits      bit (1) unal,
        3 array_info           bit (1) unal,
        3 variant_info         bit (1) unal,
        3 pad                  bit (3) unal,
      2 version                fixed bin (17) unal,
      2 type                   fixed bin (17) unal,
      2 type_offset            fixed bin (18) unsigned unal;

dcl 1 pascal_name_next         aligned based,
      2 name                   fixed bin (18) unsigned unal,
      2 next_token             fixed bin (18) unsigned unal;

dcl 1 pascal_base_type_info    aligned based,
      2 base_type              fixed bin (17) unal,
      2 base_type_offset       fixed bin (18) unsigned unal;

dcl 1 pascal_address           aligned based,
      2 location               fixed bin (18) unsigned unal,
      2 class                  fixed bin (6) unsigned unal,
      2 use_digit              bit (1) unal,
      2 units                  bit (2) unal,
      2 offset_is_encoded      bit (1) unal,
      2 pad                    bit (8) unal;

dcl 1 pascal_father_brother    aligned based,
      2 father                 fixed bin (18) unsigned unal,
      2 brother                fixed bin (18) unsigned unal;

dcl 1 pascal_son_level         aligned based,
      2 son                    fixed bin (18) unsigned unal,
      2 level                  fixed bin (6) unsigned unal,
      2 pad                    bit (12) unal;

dcl 1 pascal_father_type_successor aligned based,
      2 father_type            fixed bin (18) unsigned unal,
      2 successor              fixed bin (18) unsigned unal;

dcl pascal_size                fixed bin (35) based;

dcl pascal_offset              fixed bin (35) based;
```

```
dcl 1 pascal_subrange_limits aligned based,
      2 flags                aligned,
        3 lower_bound_is_encoded bit (1) unal,
        3 upper_bound_is_encoded bit (1) unal,
        3 pad                  bit (34) unal,
      2 subrange_lower_bound fixed bin (35),
      2 subrange_upper_bound fixed bin (35);

dcl 1 pascal_array_info      aligned based,
      2 access_info          aligned,
        3 ndims              fixed bin (6) unsigned unal,
        3 use_digit          fixed bin (1) unsigned unal,
        3 array_units        fixed bin (2) unsigned unal,
        3 virtual_origin_is_encoded bit (1) unal,
        3 pad                bit (26) unal,
      2 virtual_origin       fixed bin (35),
      2 bounds (nd refer (pascal_array_info.access_info.ndims))
                             aligned,
        3 lower              fixed bin (35),
        3 upper              fixed bin (35),
        3 multiplier         fixed bin (35),
        3 subscript_type     fixed bin (17) unal,
        3 subscript_type_offset fixed bin (18) unsigned unal,
        3 flags aligned,
          4 lower_is_encoded bit (1) unal,
          4 upper_is_encoded bit (1) unal,
          4 multipler_is_encoded bit (1) unal,
          4 pad              bit (33) unal;

dcl 1 pascal_variant_info    aligned based,
      2 number_of_variants   fixed bin (17) unal,
      2 pad                  bit (18) unal,
      2 first_value_in_set   fixed bin (35) unal,
      2 case (nvariants refer
              (pascal_variant_info.number_of_variants)),
        3 set_offset         fixed bin (18) unsigned unal,
        3 brother            fixed bin (18) unsigned unal;
```

In the discussion that follows, the term "current identifier" means the identifier represented by the symbol node under consideration, and the term "current block" means the block in which the current identifier is declared.

STRUCTURE ELEMENTS

version_flag
is "0"b to distinguish this format from the PL/I node format and to imply that the version field is valid.

aligned
is "1"b if the current identifier is aligned and is "0"b if the identifier is unaligned.

packed
is "1"b if the current identifier or type is declared packed.

in_with_block
is "1"b if the current identifier is referenced in a Pascal "with" block and is therefore not fully qualified.

name_next
is "1"b if this node contains the pascal_name_next structure.

base_type_info
is "1"b if this node contains the pascal_base_type_info structure.

address
is "1"b if this node contains the pascal_address structure.

father_brother
is "1"b if this node contains the pascal_father_brother structure.

son_level
is "1"b if this node contains the pascal_son_level structure.

father_type_successor
is "1"b if this node contains the pascal_father_type_successor structure.

size
is "1"b if this node contains the pascal_size item.

offset
is "1"b if this node contains the pascal_offset item.

subrange_limits
is "1"b if this node contains the pascal_subrange_limits structure.

array_info
is "1"b if this node contains the pascal_array_info structure.

variant_info
is "1"b if this node contains the pascal_variant_info structure.

pad
is reserved for future use and must be "0"b.

version
is 1 for this node format.

type
is the data type of the current identifier or type. The values are defined in Appendix D. Some of the values are also used in argument descriptors.

type_offset
is a self-relative pointer to a symbol node for the type that the current identifier belongs to. This component is used for Pascal user-defined and enumerated type variables, constants, record files, procedures and subscripts.

name
is a self_relative pointer to the ACC string that gives the name of the current identifier. For historical reasons, the name component points to runtime_token.name instead of to the beginning of runtime_token.

next_token
is a self-relative pointer to the symbol node of the next identifier having the same name as the current identifier.

base_type
is a Multics data type code and is used in the following cases: For subranges, it is either integer, Pascal char or Pascal enumerated type instance. For arrays, sets and record files, it is the type of the elements. For Pascal typed pointers, it is the type of the referenced variable. For function procedure types, it is the type of the return value; for other procedure types, it is not used. For non-procedure formal parameters, it is the type of the parameter.

base_type_offset
is a self-relative pointer to a symbol node describing base_type, when base_type itself is neither 0 nor a simple type.

location
usually contains a positive integer L that is used in combination with class to determine the address of the current identifier. L is normally an offset with respect to the start of a given class of storage; its interpretation depends on the value of the pascal_address.class field.

use_digit
use_digit contains the most significant bit of the three-bit binary integer that identifies the addressing units for offsets.

units
contains the low order two bits of a positive binary integer that gives the addressing units of the offset field in the symbol node. The high order bit is supplied by use_digit. The possible values and associated conversion factors are the same as for array_units in the PL/I symbol node.

offset_is_encoded
is "1"b if pascal_offset is an encoded value.

father
is a self-relative pointer to either a block node or a symbol node. If level <= 1 and the in_with_block flag is off, father points to the block node that represents the block in which the current identifier is declared. If the in_with_block flag is on, father points to the block node that represents the "block" which is the scope of the "with" statement. If level = 2, father points to the symbol node for the containing record type. If type is one of the formal parameter data types, father points to the symbol node for the procedure type that the parameter is associated with.

brother
is a self-relative pointer to the symbol node for the next identifier at the same aggregate level. If either the in_with_block flag is on or level = 2 and the variant_info flag is off, brother points to the symbol node for the next field in the record. If the variant_info flag is on, brother is 0 (undefined) since the brother fields in pascal_variant_info are used instead. If type is one of the formal parameter data types, brother points to the symbol node for the procedure's next parameter. Otherwise, if level <= 1 and the name_next flag is on, brother points to the next element on the list of symbol nodes ordered alphabetically by the size of the name.

son
is a self-relative pointer. If type = Pascal record type, son points to the symbol node for the first field in the record. If type = Pascal enumerated type, son points to the symbol node for the first constant of the type. If type = Pascal procedure type, son points to the symbol node for the first formal parameter.

level
is 1 if type = Pascal record type. level is 2 in symbol nodes representing record fields, except if the in_with_block flag is on. Otherwise, level is 0 or nonexistent.

father_type
is a self-relative pointer to the symbol node for the containing enumerated type if type = Pascal enumerated type element.

successor
is a self-relative pointer to the symbol node for the next element in the set of enumerated values for the containing enumerated type if type = Pascal enumerated type element.

pascal_size
is the arithmetic precision for numeric types (integer, real, integer subrange). It is the length in bits for everything else. For char, enumerated type instance and enumerated type element, it can be considered as the arithmetic precision of the unsigned internal code.

pascal_offset
is the offset of a record field with respect to the beginning of the record if the in_with_block flag is off and level = 2. If the in_with_block flag is on, pascal_offset is the offset of the record field with respect to the address specified by location and class. It is encoded if offset_is_encoded = "1"b.

lower_bound_is_encoded
is "1"b if subrange_lower_bound is an encoded value.

upper_bound_is_encoded
is "1"b if subrange_upper_bound is an encoded value.

subrange_lower_bound
is the lower bound of the subrange. If it is not encoded, it contains an integer or Pascal char value.

subrange_upper_bound
is the upper bound of the subrange. If it is not encoded, it contains an integer or Pascal char value.

ndims
is the number of array dimensions.

use_digit
contains the most significant bit of the three bit binary integer that identifies the addressing units for the array.

array_units
contains the low order two bits of a three bit positive binary integer that gives the addressing units to be used when computing the address of an array element. The high order bit is supplied by use_digit. The possible values and associated conversion factors are the same as for array_units in the PL/I symbol node.

virtual_origin_is_encoded
is "1"b if virtual_origin is an encoded value.

virtual_origin
is the value of the virtual origin of the array, in units given by array_units. It should be subtracted from the base address specified by the location and class of an array variable. This field is not used when the array is conformant, i.e. when bounds(1).flags.lower_is_encoded = "1"b. In this case, the virtual origin is computed as the sum of array_info.bounds(i).lower * array_info.bounds(i).multiplier for i = 1 to ndims.

lower
is the lower bound of this dimension of the array.

upper
is the upper bound of this dimension of the array.

multiplier
is the multiplier of this dimension of the array.

subscript_type
is the data type of this dimension's subscript. It is zero if the subscript is numeric and not previously declared, i.e. if a numeric subrange is given explicitly in the array declaration. subscript_type_offset
is a relative pointer to a symbol node describing subscript_type.

lower_is_encoded
is "1"b if bounds(i).lower is an encoded value.

upper_is_encoded
is "1"b if bounds(i).upper is an encoded value.

multiplier_is_encoded
is "1"b if bounds(i).multiplier is an encoded value.

number_of_variants
is the number of variants in the containing record.

first_value_in_set
is the lowest value used to select a variant.

set_offset
is a self-relative pointer to a bit string that specifies the cases of the variant. The bit string represents a set (one bit per set element) whose base type is the type of the current symbol node. The first bit corresponds to first_value_in_set.

brother
is a self_relative pointer to the first field of the variant part.

Additional Information About Pascal Symbol Nodes

The following table summarizes which structure items are present in the Pascal symbol node for each type of data element. Columns correspond to distinct Pascal data types. Rows correspond to the fields in the symbol node format declared above. The table shows an X where the corresponding field is simply present. Other symbols give more detailed information as listed in the accompanying key.

Table G-1. Contents of Pascal Symbol Nodes

Type Nodes

Item	1	2	3	4	5	6	7	8	9
type	73	73	73	70	64	69	68	67	76
type_offset	0	0	0	0	0	0	0	0	0
name	X*	X*	X*	X*	X*	X*	X*	X*	
next_token	X*	X*	X*	X*	X*	X*	X*	X*	
base_type	1,65	A	72		C	X		X	D
base_type_offset	0	X	X		X	X		X	G
address_info									
father	X*	X*	X*	X*	X*	X*	X*	X*	
brother	X*	X*	X*	X*	X*	X*	X*	X*	
son				B			X		E
level				0			1		0
father_type									
successor									
size		X			X				F
offset									
subrange limits	X		X						
array info		X							
variant info									

1 integer or char subrange
2 array
3 enumerated type subrange
4 Pascal enumerated type
5 Pascal typed pointer type
6 Pascal set type
7 Pascal record type
8 Pascal record file type
9 Pascal procedure type

A type code for elements of the array
B offset of symbol node for the first constant of the type
C type code for the referenced variable (any variable type)
D type of the returned value (only if procedure is a function)
E offset of the symbol node for the first formal parameter
F size corresponding to the type of the returned value (if procedure is a function)
G only for function procedure types

X* field exists if type has a name

Table G-1. Contents of Pascal Symbol Nodes (Continued)

item	10	11	12	13	14	15	16	17	18	19	20
type	X	X	X	X	W	71	74	24	Y	Z	79
type_offset	X	X	X	0	0	0	Q	0	S	0	0
name	X	X	X*	X	X	X	X	X	X	X	X
next_token	X	X	X*	X	X	X	X	X	X	0	0
base_type										X	
base_type_offset										X	
location	X	0	0	X	X	X	X	X	X		
class	X	0	0	X	X	X	X	X	X		
units	X	X	X	X							
father	X	H	H	J	X	X	X	X	X	S	S
brother	X	I	0	K	X	X	X	X	X	V	V
son		0	0								
level		2	2								
father_type						X					
successor						X					
size	X	X	X*	X	L	P	R		T	X	X
offset		X	X*	X							
subrange limits											
array info											
variant info			X								

10 variables
11 simple fields in records
12 selector (tag) fields in records
13 record fields accessed in "with" blocks; in_with_block flag on
14 integer, real and char constants
15 enumerated type elements (constants)
16 string constants
17 labels
18 procedures
19 nonprocedure formal parameters
20 procedure formal parameters

H offset of symbol node for containing record type
I offset of symbol node for next field (0 if last)
J offset of "with" block node
K offset of brother field node in "with" block
L size in bits (for char); precision of generated constant (for integer and real)
P size in bits of internal code
Q offset of symbol node for type "packed array [1..number_of_chars] of char"
R length in bits (9 * number_of_chars)
S offset of a procedure type symbol node
T only for pascal procedure parameter; =number of words in datum
V offset of symbol node for the next formal parameter; 0 if last
W types 1, 4, 65
Y types 25, 26, 27, 80
Z types 77, 78

One of the major differences between Pascal and PL/I symbol tables, aside from the node format itself, is that in Pascal it is necessary to describe types themselves in symbol nodes. The following table shows the relationship between the data type codes used by variables and those used by types.

Table G-2. Data Type Codes Used by Variables vs. Types

type codes used for variables, record fields, constants, subscript types and base types	*corresponding type_offset and type codes used for types*
(1) integer	predefined (no type_offset)
(4) real	"
(65) pascal char	"
(66) pascal boolean	"
(75) pascal text file	type_offset is the relative
(72) pascal enumerated type instance	offset of a symbol node with type code enumerated type (70)
(74) pascal user defined type instance	type_offset is the relative offset of a symbol node with one of the following type codes: (73) pascal user_defined type (used for arrays and subranges) (67) pascal record file type (64) pascal typed pointer type (68) pascal record type (69) pascal set type

SPECIAL RUNTIME SYMBOL DATA TYPE CODES

type	*data type*
24	label constant (used in symbol tables only)
25	internal entry constant (used in symbol tables only)
26	external entry constant (used in symbol tables only)
27	external procedure (used in symbol tables only)
63	picture (used in symbol tables only)
70	Pascal enumerated type
71	Pascal enumerated type element
73	Pascal user defined type
77	Pascal variable formal parameter
78	Pascal value formal parameter
79	Pascal entry formal parameter
80	Pascal parameter procedure

These types are used in runtime_symbol values only, and not in argument descriptors. The user is referred to std_descriptor_types.incl.pl1, which gives named constants for these codes. See Appendix D for more information.

The Statement Map

The statement map contains information about each statement in the source program for which instructions were generated. The statement map is normally placed after the runtime symbol table, if the table is present. All the entries are contiguous. Each entry in the statement map has the following format (declared in statement_map.incl.pl1):

```
dcl 1 statement_map      aligned based,
      2 location         bit(18) unaligned,
      2 source_id        unaligned,
        3 file           bit(8),
        3 line           bit(14),
        3 statement      bit(5),
      2 source_info      unaligned,
        3 start          bit(18),
        3 length         bit(9);
```

STRUCTURE ELEMENTS

location
is location in the object segment of the first instruction generated for the statement that corresponds to this entry in the statement

source_id
describes the line on which the statement begins. The last entry in the statement map is a dummy that has string(source_id) = (27)"1"b.

file
contains a positive binary integer that specifies the number of the source segment in which the current statement is contained (see "The Source Map").

line
contains a positive binary integer that specifies the number of the line on which the current statement begins. The first line in a file is number 1.

statement
contains a positive binary integer that specifies the position of the current statement on the line in which it begins. The first statement on a line is number 1.

source_info
: specifies the starting position and length of the string of characters that are the source for the current statement.

start
: contains a positive binary integer S that specifies the number of characters that precede the first character of the source of the current statement (see below).

length
: contains a positive binary integer L that gives the number of characters occupied by the current statement in the source file; a statement is assumed to be entirely contained in a single segment. If string is the contents of the source file that contains the current statement considered as a single string, the source string for the current statement is substr(string,S+1,L).

APPENDIX H

STANDARD EXECUTION ENVIRONMENT

STANDARD STACK AND LINK AREA FORMATS

Because of the linkage mechanism, stack manipulations, and the complexity of the Multics hardware, a series of Multics execution environment standards have been adopted. All standard translators (including assemblers) adhere to these standards as do all supervisor and standard storage system procedures. Furthermore, they assume that other procedures do so as well.

Multics Stack

The normal mode of execution in a standard Multics process uses a stack segment. There is one stack segment for each ring. The stack for a given ring has the entryname stack_R, where R is the ring number, and is located in the process directory. Each stack contains a "header" followed by as many "stack frames" as are required by the executing procedures. A stack header contains pointers to special code and data that are initialized when the stack is created. Some of these pointers are variable and change during process execution. They are included in the stack header so that they can always be retrieved without supervisor intervention (for efficiency). The actual format of the stack header is described under "Stack Header" below.

Stack frames begin at a location specified in the stack header, are variable in length, and contain both control information and data for dynamically active procedures. In general, a stack frame is allocated by the procedure to which it belongs when that procedure is invoked. The stack frames are threaded to each other with forward and backward pointers, making it an easy task to trace the stack in either direction. The stack usage described below is critical to normal Multics operation; any deviations from the stated discipline can result in unexpected behavior.

Stack Header

The stack header contains pointers (on a per-ring basis) to information about the process, to operator segments, and to code sequences that can be used to invoke the standard call, push, pop, and return functions (described below). Figure H-1 gives the format of the stack header. The following descriptions are based on that figure and on the following PL/I declaration.

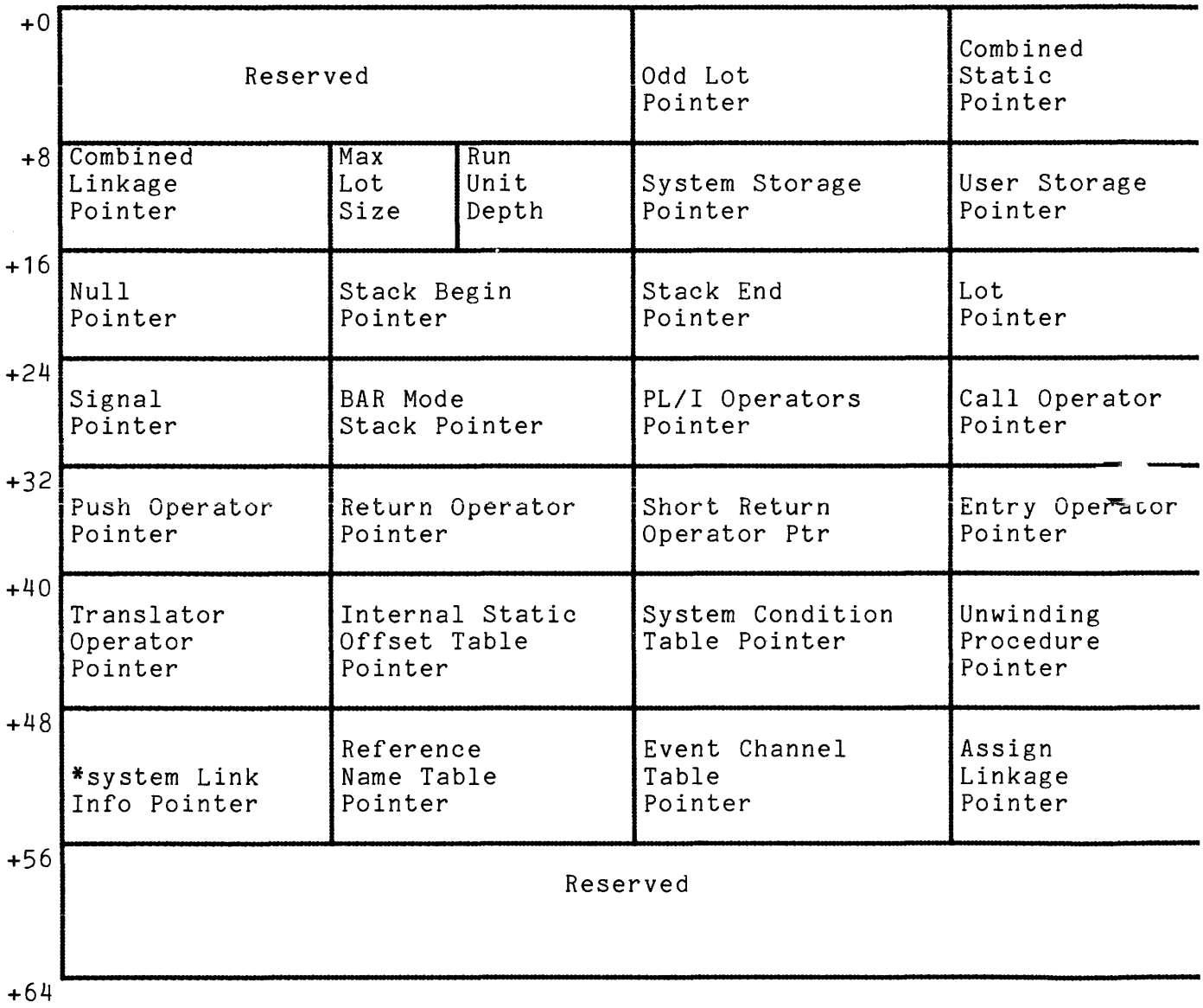


Figure H-1. Stack Header Format

```
dcl 1 stack_header              based aligned,
      2 pad1 (4)                fixed bin,
      2 old_lot_ptr             ptr,
      2 combined_stat_ptr       ptr,
      2 clr_ptr                 ptr,
      2 max_lot_size            fixed bin(17) unaligned,
      2 main_proc_invoked       fixed bin (11) unaligned,
      2 have_static_vlas        bit(1) unaligned,
      2 pad4                    bit(2) unaligned,
      2 run_unit_depth          fixed bin(2) unaligned,
      2 cur_lot_size            fixed bin(17) unaligned,
      2 pad2                    bit(18) unaligned,
      2 system_storage_ptr      ptr,
      2 user_storage_ptr        ptr,
      2 null_ptr                ptr,
      2 stack_begin_ptr         ptr,
      2 stack_end_ptr           ptr,
      2 lot_ptr                 ptr,
      2 signal_ptr              ptr,
      2 bar_mode_sp_ptr         ptr,
      2 pl1_operators_ptr       ptr,
      2 call_op_ptr             ptr,
      2 push_op_ptr             ptr,
      2 return_op_ptr           ptr,
      2 short_return_op_ptr     ptr,
      2 entry_op_ptr            ptr,
      2 trans_op_tv_ptr         ptr,
      2 isot_ptr                ptr,
      2 sct_ptr                 ptr,
      2 unwinder_ptr            ptr,
      2 sys_link_info_ptr       ptr,
      2 rnt_ptr                 ptr,
      2 ect_ptr                 ptr,
      2 assign_linkage_ptr      ptr,
      2 reserved(5)             bit(36) aligned,
      2 pad3                    bit(36) aligned;
```

STRUCTURE ELEMENTS

pad1
is unused.

old_lot_ptr
is a pointer to the linkage offset table (LOT) for the current ring. This field is obsolete.

combined_stat_ptr
is a pointer to the area in which separate static sections are allocated.

clr_ptr
is a pointer to the area in which linkage sections are allocated.

max_lot_size
is the maximum number of words (entries) that the LOT and internal static offset table (ISOT) can have.

main_proc_invoked
is nonzero if a main procedure was invoked in a currently active run unit.

have_static_vlas
is "1"b if internal static large or very large arrays are being used.

pad4
is unused and must be "0"b.

run_unit_depth
is the current run unit level.

cur_lot_size
is the current number of words (entries) in the LOT and ISOT.

pad2
is unused.

system_storage_ptr
is a pointer to the area used for system storage, which includes command storage and the *system link name table.

user_storage_ptr
is a pointer to the area used for user storage, which includes FORTRAN common and PL/I external static variables whose names do not include "$".

null_ptr
contains a null pointer value. In some circumstances, the stack header can be treated as a stack frame. When this is done, the null pointer field occupies the same location as the previous stack frame pointer of the stack frame. (See "Multics Stack Frame" below.) A null pointer indicates that there is no stack frame prior to the current one.

stack_begin_ptr
is a pointer to the first stack frame on the stack. The first stack frame does not necessarily begin at the end of the stack header. Other information, such as the linkage offset table, can be located between the stack header and the first stack frame.

stack_end_ptr
: is a pointer to the first unused word after the last stack frame. It points to the location where the next stack frame is placed on this stack (if one is needed). A stack frame must be a multiple of 16 words; thus, both of the above pointers point to 0 (mod 16) word boundaries.

lot_ptr
: is a pointer to the linkage offset table (LOT) for the current ring. The LOT contains packed pointers to the dynamic linkage sections known in the ring in which the LOT exists. The linkage offset table is described below under "Linkage Offset Table."

signal_ptr
: is a pointer to the signalling procedure to be invoked when a condition is raised in the current ring.

bar_mode_sp_ptr
: is a pointer to the stack frame in effect when BAR mode was entered. (This is needed because typical BAR mode programs can change the word offset of the stack frame pointer register.)

pl1_operators_ptr
: is a pointer to the standard operator segment used by PL/I. It is used by PL/I and FORTRAN object code to locate the appropriate operator segment.

call_op_ptr
: is a pointer to the Multics standard call operator used by ALM procedures. It is used to invoke another procedure in the standard way.

push_op_ptr
: is a pointer to the Multics standard push operator that is used by ALM programs when allocating a new stack frame. All push operations performed on a Multics stack should use either this or an equivalent operator; otherwise results are unpredictable. (The push operation was formerly called save.)

return_op_ptr
: is a pointer to the Multics standard return operator used by ALM procedures. It assumes that a push has been performed by the invoking ALM procedure and pops the stack prior to returning control to the caller of the ALM procedure.

short_return_op_ptr
: is a pointer to the Multics standard short return operator used by ALM procedures. It is invoked by a procedure that has not performed a push to return control to its caller.

entry_op_ptr
is a pointer to the Multics standard entry operator. The entry operator does little more than find a pointer to the invoker's linkage section.

isot_ptr
points to a vector of pointers to special language operators; this table can be expanded to accommodate new languages without causing a change in the stack header.

isot_ptr
is a pointer to the internal static offset table (ISOT). The ISOT contains packed pointers to the dynamic internal static sections known in the ring in which the ISOT exists.

sct_ptr
is a pointer to the system condition table (SCT) used by system code in handling certain events.

unwinder_ptr
is a pointer to the unwinding procedure to be invoked when a nonlocal goto is executed in the current ring.

sys_link_info_ptr
is a pointer to the *system link name table.

rnt_ptr
points to the reference name table (RNT).

ect_ptr
points to the event channel table (ECT).

assign_linkage_ptr
points to the area used by certain critical system programs whose operations must not be modified by run unit. This pointer initially points to the same area as stack_header.clr_ptr but is not changed by the run unit mechanism.

reserved
is reserved.

pad3
is unused.

The call, push, return, short_return, and entry operators are invoked by the object code generated by the ALM assembler. Other translators that intend to use the standard call/push/return strategy should either use these operators or an operator segment with a set of operators consistent with these. For a detailed description of what the operators do and how to invoke them, see "Subroutine Calling Sequences" later in this section.

The PL/I and FORTRAN compilers use slightly different operators that perform equivalent and compatible functions. All supported translators, however, depend on the effects generated by these operators.

Multics Stack Frame

The format given below for a standard Multics stack frame must be strictly followed because several critical procedures of the Multics system depend on it. A bad stack segment or stack frame can easily lead to process termination, looping, and other undesirable effects.

In the discussion that follows, the "owner" of a stack frame is the procedure that created it (with a push operation). Some programs (generally ALM programs) never perform a push and hence do not own a stack frame. If a procedure that does not own a stack frame is executing, it can neither call another procedure nor use stack temporaries; all stack information refers to the program that called such a program.

Figure H-2 illustrates the detailed structure of a stack frame (the standard use in ALM). The following descriptions are based on that diagram and on the following PL/I declaration.

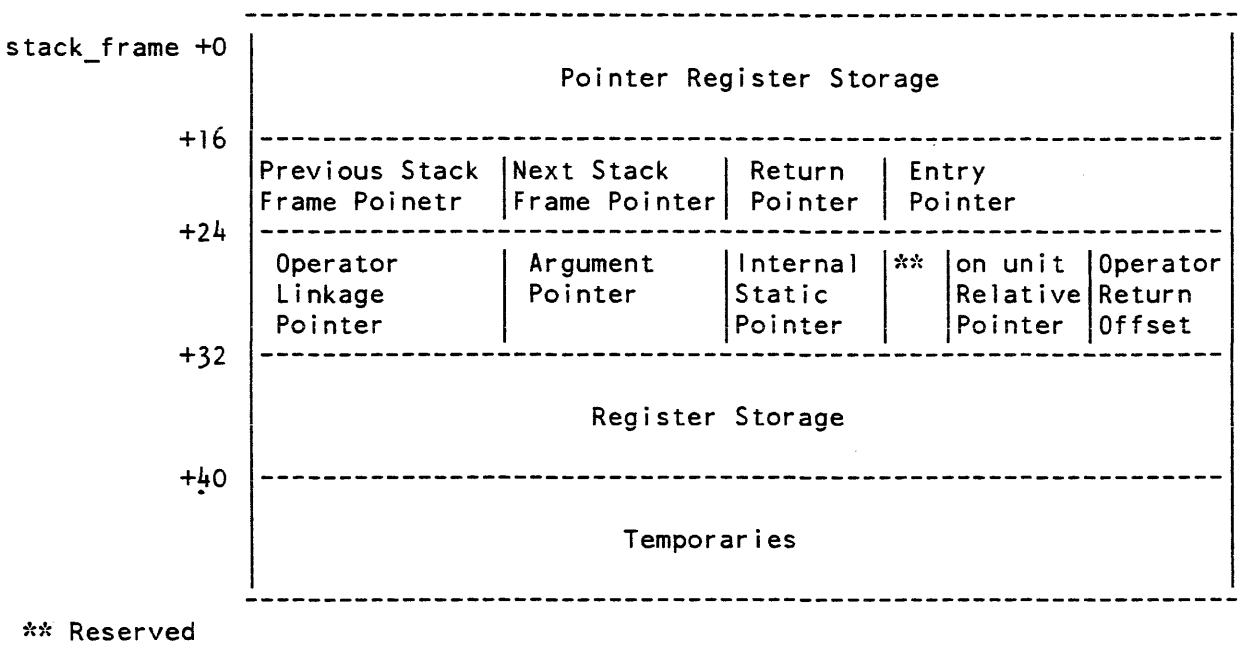


Figure H-2. Stack Frame Format

```
dcl 1 stack_frame                based (sp) aligned,
      2 prs(16)                  fixed bin,
      2 prev_stack_frame_ptr     ptr,
      2 next_stack_frame_ptr     ptr,
      2 return_ptr               ptr,
      2 entry_ptr                ptr,
      2 operator_link_ptr        ptr,
      2 argument_ptr             ptr,
      2 static_ptr               ptr unaligned,
      2 reserved                 fixed bin,
      2 on_unit_rel_ptrs(2)      bit(18) unaligned,
      2 translator_id            bit(18) unaligned,
      2 operator_return_offset   bit(18) unaligned,
      2 regs(8)                  fixed bin;
```

prs
: is used to save pointer registers of the calling program when the ALM call operator is invoked.

prev_stack_frame_ptr
: is a pointer to the base of the stack frame of the procedure that called the procedure owning the current stack frame. This pointer may or may not point to a stack frame in the same stack segment.

next_stack_frame_ptr
: a pointer to the base of the next stack frame. For the last stack frame on a stack, the pointer points to the next available area in the stack where a procedure can lay down a stack frame; i.e., it has the same value as the stack_end_ptr in the stack header. The previous stack frame pointers and the next stack frame pointers form threads through all active frames on the stack. These two threads are used by debugging tools to search and trace the stack as well as by the call/push/return mechanism.

return_ptr
: is a pointer to the location to which a return can be made in the procedure that owns the given frame. This pointer is undefined if the procedure has never made an external call, and points to the return location associated with the last external call if the given procedure has been returned to and is currently executing.

entry_ptr
: is a pointer to the procedure entry point that was called and that owns the stack frame. The pointer points to a standard entry point. See "Structure of the Text Section" in Appendix G.

operator_link_ptr
: is usually the operator pointer being used by the procedure that owns the given stack frame. For ALM programs, this points to the linkage section of the procedure.

argument_ptr
: is a pointer to the argument list passed to the procedure that owns the given stack frame.

static_ptr
: is a pointer to the internal static storage for the procedure owning the stack frame.

reserved
: is reserved for future use. 0..kx on_unit_rel_ptrs is a pair of relative pointers to on unit information contained within the stack frame. This on unit information is valid only if bit 29 of the first word of prev_stack_frame_ptr is a 1. (This bit is automatically set to 0 when a push is performed by the procedure that owns the stack frame.) The first of the on_unit_rel_ptrs is a pointer (relative to the stack frame base) to a list of enabled conditions. The second of the on_unit_rel_ptrs is obsolete.

translator_id
is a coded number indicating the translator used to generate the object code of the owner of the stack frame.

operator_return_offset
operator_return_offset contains a return location for certain pl1_operators_ functions. If it is nonzero, it is a relative pointer to the return location in the compiled program (return from pl1_operators_). If it is zero, a dedicated register (known by pl1_operators_) contains the return location.

regs
is used to save arithmetic registers of the calling program when the ALM call operator is invoked.

Two major areas of a stack frame not explicitly defined above are the first 16 words and words 32 through 39. The contents of these areas is not always defined or meaningful, although they have a well-defined purpose for ALM programs and are used internally by the PL/I and FORTRAN programs. The procedure owning the stack frame can use these areas as it sees fit.

Linkage Offset Table

As described above, each stack header contains a pointer to the linkage offset table (LOT) for the current ring. The LOT is an array, indexed by text segment number, of packed pointers to the linkage sections for the procedure segments known in the current ring.

The structure of the LOT is defined by the following PL/I declaration:

```
dcl 1 lot based (lot_ptr)                                aligned,
      2 linkage_ptr (0: stack_header.cur_lot_size-1) ptr unaligned;
```

where linkage_ptr is the array of linkage section pointers.

If one of the slots in the linkage_ptr array contains all 0's, the segment number associated with the slot does not correspond to a known segment.

If one of the slots in the linkage_ptr array contains all 0's except for "111"b in the high-order three bits (a lot fault), the segment number associated with the slot corresponds to a known segment that either does not have a linkage section or whose linkage section has not been combined (i.e., the segment has not been executed).

Internal Static Offset Table

The stack header in each ring contains a pointer to the internal static offset table (ISOT) for the current ring. The ISOT is an array, indexed by text segment number, of packed pointers to the internal static sections for the corresponding procedure segments known in the current ring. Since the ISOT always immediately follows the LOT, the isot_ptr is redundant but is retained for efficiency.

The internal static pointers are identical to the linkage section pointers unless the corresponding object segment was generated with separate static. If the static is separate, i.e., not allocated in the linkage section, the internal static pointer either points to the allocated static or contains a value that causes an "isot fault" if referenced.

The structure of the ISOT is defined by the following PL/I declaration:

```
dcl 1 isot based (isot_ptr) aligned,
      2 static_ptr (0: stack_header.cur_lot_size-1)  ptr unaligned;
```

where static_ptr is the array of static/linkage section pointers.

SUBROUTINE CALLING SEQUENCES

The Multics standard call and return conventions are described in the following paragraphs. For information about the format of stack segments and stack frames, see "Standard Stack and Linkage Area Formats" above.

The call and return from one procedure to another can be broken down into seven separate steps. Operators to perform these steps have been provided in the standard operator segment named pl1_operators_ (for PL/I, FORTRAN, and ALM procedures). These operators are invoked when appropriate by the object code generated by these translators.

The steps involved in a call and return and the associated operators are listed below.

1. A procedure call, i.e., a transfer of control and passing of an argument list pointer to the called procedure (call).

2. Generation of a linkage (and internal static) pointer for the called procedure (entry).

3. Creation of a stack frame for the called procedure (push).

4. Storage of standard items to be saved in the stack frame of the called procedure (entry and push).

5. Release of the stack frame of the called procedure just prior to returning (return).

6. Reestablishment of the execution environment of the calling procedure (return and short_return).

7. Return of control to the calling procedure (return and short_return).

Preparation of the argument list, although necessary, was not listed above because the operators need know nothing about the format of an argument list. See "Argument List Format" later in this section.

The following description is based on the operators used by ALM procedures. The operators used by PL/I and FORTRAN procedures are basically the same but differ at a detailed level due to: (1) slight changes in the execution environment when PL/I and FORTRAN programs are running; and (2) simplification and combination of operators made possible by the execution environment of PL/I. The PL/I and FORTRAN operators are not described here other than to define a minimum execution environment that must be established when returning to a PL/I or FORTRAN program.

(The following description is given in terms of Honeywell hardware.)

Call Operator

The call operator transfers control to the called procedure. This operator is invoked in two ways from ALM procedures. The first is a result of the call pseudo-op, which invokes the call operator after saving the machine registers in the calling program's stack frame and loading pointer register 0 with a pointer to the argument list to be passed to the called procedure. Upon return to the calling program, these saved values are restored into the hardware registers by the calling procedure. The second way that ALM procedures can invoke the call operator is through the short_call pseudo-op. This is used when the calling procedure does not need all of the machine registers saved and restored across the call. The ALM procedure can selectively save whatever registers are needed.

Neither the call nor the short_call pseudo-ops (nor the PL/I and FORTRAN equivalents) require or expect the machine registers to be restored by the called procedure. In fact, only the pointer registers 0 (operator segment pointer) and 6 (stack frame pointer) are ever guaranteed to be restored across a call. It is up to the calling procedure to save and restore any other machine registers that are needed.

Entry Operator

The entry operator used by ALM programs performs two functions. It generates a pointer to the linkage section of the called procedure (which it leaves in pointer register 4) and it stores a pointer to the entry in what will be the stack frame of the called procedure (if the procedure ever creates a stack frame for itself). At the time the entry operator is invoked, a new stack frame has not yet been established. Indeed, the called procedure may never create one. However, it is certainly possible to know where the stack frame will go if and when it is created and this knowledge is used to store the entry pointer.

The entry operator is invoked by an ALM procedure that transfers to a label in another procedure that has been declared as an entry through the entry pseudo-op. The transfer is made to a standard entry structure the first executable word of which is (PR7 is assumed to point to the base of the current stack segment):

```
tsp2 7|entry_op,*
```

The operator returns to the instruction after the tsp2 instruction, which may or may not be another transfer instruction. (A link to the entry, when snapped, points to the tsp2 instruction.) See "Structure of the Text Section" in Appendix G.

Some ALM programs may not require a linkage pointer. Such programs can declare the label to which control should be transferred with a segdef pseudo-op. This causes the appropriate definition and linkage information to be generated so that other procedures can find the entry point. When called, the transfer is straight to the code at the label and the normal entry structure is not generated or used. No linkage pointer is found and no entry pointer is saved. This technique is recommended only where speed of execution is of utmost importance since it avoids calculation of useful diagnostic information.

Push Operator

The push operator used by ALM procedures is invoked as a result of the push pseudo-op that is used to create a stack frame for the called procedure. In addition to creating a stack frame, several pointers are saved in the new stack frame. They are:

- Argument pointer
- Linkage pointer (and internal static pointer)
- Previous stack frame pointer
- Next stack frame pointer

If the called procedure is defined as an entry (rather than segdef), the entry pointer has already been saved in the new stack frame.

The push pseudo-op must be invoked if the called procedure makes further calls itself or uses temporary storage. Due to their manner of execution, PL/I and FORTRAN procedures combine the entry and push operators into a single operator.

The push operator and the return operators are managers of the stack frames and the stack segment in general. The push operator establishes the forward and backward stack frame threads and updates the stack end pointer in the stack header appropriately. The return operators use these threads and also update the stack end pointer as needed. Any program that wishes to duplicate these functions must do so in a way that is compatible with the procedures outlined in this discussion and those described above under the heading "Standard Stack and Linkage Area Formats."

Return Operator

The return operator is invoked by ALM procedures that have specified the return pseudo-op. The return operator pops the stack, reestablishes the minimum execution environment, and returns control to the calling procedure. The only registers restored are pointer registers 0 and 6, as mentioned above.

Short Return Operator

The short_return operator is invoked by ALM procedures that have specified the short_return pseudo-op. The short_return operator differs from the return operator in that the stack frame is not popped. This return is used by ALM procedures that did not perform a push.

Pseudo-op Code Sequences

The following code sequences are generated by the assembler for the specified pseudo-op.

```
call:

     OBJECT CODE      spri     pr6|0
                      sreg     pr6|32
                      epp0     arglist
                      epp2     entrypoint
                      tsp4     pr7|stack_header.call_op,*

     OPERATORS        spri4    pr6|stack_frame.return_ptr
                      sti      pr6|stack_frame.return_ptr+1
                      epp4     pr6|stack_frame.lp_ptr,*
                      call6    pr2|0

     OBJECT CODE      lpri     pr6|0
                      lreg     pr6|32

short_call:

     OBJECT CODE      epp2     entrypoint
                      tsp4     pr7|stack_header.call_op,*

     OPERATORS        (as above)

     OBJECT CODE      epp4     pr6|stack_frame.lp_ptr,*


return:

     OBJECT CODE      tra      pr7|stack_header.return_op,*

     OPERATORS        spri6    pr7|stack_header.stack_end_ptr
                      epp6     pr6|stack_frame.prev_sp,*
                      epbp7    pr6|0
                      epp0     pr6|stack_frame.operator_ptr,*
                      ldi      pr6|stack_frame.return_ptr+1
                      rtcd     pr6|stack_frame.return_ptr
```

```
short_return:

     OBJECT CODE      tra      pr7|stack_header.short_return_op,*


     OPERATORS        epbp7    pr6|0
                      epp0     pr6|stack_frame.operator_ptr,*
                      ldi      pr6|stack_frame.return_ptr+1
                      rtcd     pr6|stack_frame.return_ptr

entry:

     OBJECT CODE      tsp2     pr7|stack_header.entry_op,*

     OPERATORS        epp2     pr2|-1
                      epp4     pr7|stack_header.stack_end_ptr,*
                      spri2    pr4|stack_frame.entry_ptr
                      epaq     pr2|0
                      lprp5    pr7|stack_header.isot_ptr,*au
                      sprp5    pr4|stack_frame.static_ptr
                      lprp4    pr7|stack_header.lot_ptr,*au
                      tra      pr2|1
     OBJECT CODE      tra      executable_code

push:

     OBJECT CODE      eax7     stack_frame_size
                      tsp2     pr7|stack_header.push_op,*

     OPERATORS        spri2    pr7|stack_header.stack_end_ptr,*
                      epp2     pr7|stack_header.stack_end_ptr,*
                      spri6    pr2|stack_frame.prev_sp
                      spri0    pr2|stack_frame.arg_ptr
                      spri4    pr2|stack_frame.lp_ptr
                      epp6     pr2|0
                      epp2     pr6|0,7
                      spri2    pr7|stack_header.stack_end_ptr
                      spri2    pr6|stack_frame.next_sp
                      eax7     1
                      stx7     pr6|stack_frame.translator_id
                      tra      pr6|0,*
```

Register Usage Conventions

The following conventions, used in the standard environment, should be followed by any user-written translator.

- The only registers that are restored across a call are the pointer registers:

  0 (ap) operator segment pointer
  6 (sp) stack frame pointer

  The operator segment pointer is restored correctly only if it is saved at some time prior to the call (e.g., at entry time).

- The code generated by the ALM assembler assumes that pointer register 4 (lp) always points to the linkage section for the executing procedure and that pointer register 7(sb) always points to the stack header.

- Pointer register 7 is assumed to be pointing to the base of the stack when control is passed to a called procedure.

Argument List Format

When a standard call is performed, the argument pointer (pointer register 0) is set to point at the argument list to be used by the called procedure. The argument list must begin on an even word boundary. Its format is given by the following PL/I declaration (arg_list.incl.pl1):

```
dcl 1 arg_list                  aligned based,
      2 arg_count               fixed bin(17) unsigned unal,
      2 pad1                    bit(1) unal,
      2 call_type               fixed bin(18) unsigned unal,
      2 desc_count              fixed bin(17) unsigned unal,
      2 pad2                    bit(19) unal,
      2 arg_ptrs                (arg_list_arg_count) ptr,
      2 desc_ptrs               (arg_list_arg_count) ptr;

dcl 1 arg_list_with_envptr      aligned based,
      2 arg_count               fixed bin(17) unsigned unal,
      2 pad1                    bit(1) unal,
      2 call_type               fixed bin(18) unsigned unal,
      2 desc_count              fixed bin(17) unsigned unal,
      2 pad2                    bit(19) unal,
      2 arg_ptrs                (arg_list_arg_count) ptr,
      2 envptr                  ptr,
      2 desc_ptrs               (arg_list_arg_count) ptr;
```

```
dcl 1 command_name_arglist      aligned based,
      2 header,
        3 arg_count             fixed bin(17) unsigned unaligned,
        3 pad1                  bit(1) unaligned,
        3 call_type             fixed bin(18) unsigned unaligned,
        3 desc_count            fixed bin(17) unsigned unaligned,
        3 mbz                   bit(1) unaligned,
        3 has_command_name      bit(1) unaligned,
        3 pad2                  bit(17) unaligned,
      2 arg_ptrs                (arg_list_arg_count refer
                                  (command_name_arglist_arg_count))
                                  ptr,
      2 desc_ptrs               (arg_list_arg_count refer
                                  (command_name_arglist_arg_count))
                                  ptr,
      2 name,
        3 command_name_ptr      pointer,
        3 command_name_length   fixed bin(21);
```

This page intentionally left blank.

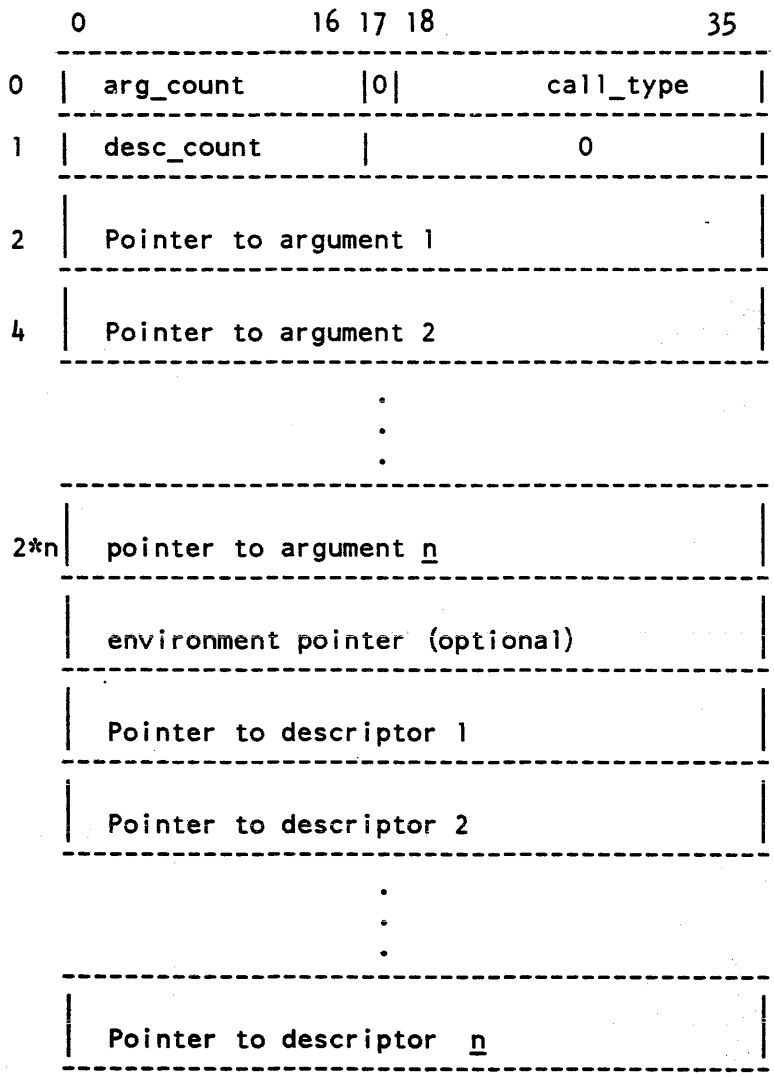


Figure H-1. Standard Argument List

arg_count
is the number of arguments passed.

pad1
is reserved and must be "0"b.

call_type
is a code that describes the type of call being made. It can have one of the following values:

```
0    for quick internal calls.
4    for inter-segment calls.
8    for calls where an environment pointer is passed.
```

The include file declares constants with these values:

```
dcl (
    Quick_call_type             init(0),
    Interseg_call_type          init(4),
    Envptr_supplied_call_type   init(8),
    )                           fixed bin(18) unsigned unal
                                     int static options
                                     (constant);
```

desc_count
is the number of argument descriptors being passed. If non-zero, it must be the same as arg_count.

pad2
is reserved and must be "0"b.

arg_ptrs
is an array of pointers to the arguments.

envptr
is the environment pointer for the procedure being called. It is present only if call_type is 8.

desc_ptrs
is an array of pointers to the argument descriptors, if present.

has_command_name
if "1"b this specifies that the command name has been stored in name_ptr and its length has been stored in name_length.

command_name_ptr
is a pointer to the expanded command name given on the command line.

command_name_length
is the length of the expanded command name pointed to by name_ptr.

NOTES: The pointers in the argument list need not be ITS pointers; however they must be pointers through which the hardware can perform indirect addressing. Packed (unaligned) pointers cannot be used.

This page intentionally left blank.

The pointer envptr is present when a call is made to a non-quick internal procedure or when a call is made through an entry variable, regardless of whether the procedure being called is an external or internal procedure. When the called procedure is an internal procedure, envptr points to a stack frame of the activation of the block that contains the called procedure, and is used to set up the display pointer for the stack frame that the non-quick procedure will create. If the call is made through an entry variable, envptr is copied from the environment ptr of the entry variable. (See elsewhere in this manual for the format of an entry variable.) If the call is to an internal entry constant, envptr is calculated by the PL/I operators. If a call is made through an entry variable to an external procedure, the environment pointer of the entry variable will be null, thus envptr is also null.

The include file also contains symbolic names for the values that call_type takes on. They are: Quick_call_type, Interseg_call_type, and Envptr_supplied_call_type.

In the include file, the extent of the arrays, arg_ptrs, and desc_ptrs is determined by the variable arg_list_arg_count (which is not declared in the include file). In references to an already allocated argument list, the programmer should first set arg_list_arg_count to the value of arg_count in the appropriate structure (arg_list or arg_list_with_envptr).

An argument pointer points directly to an argument. A descriptor pointer points to the descriptor associated with the argument.

The format of an argument descriptor is described by one of the following PL/I declarations, given in arg_descriptor.incl.pl1.

```
dcl 1 arg_descriptor          based aligned,
      2 flag                  bit(1) unal,
      2 type                  fixed bin(6) unsigned unal,
      2 packed                bit(1) unal,
      2 number_dims           fixed bin(4) unsigned unal,
      2 size                  fixed bin(24) unsigned unal;

dcl 1 fixed_arg_descriptor    based aligned,
      2 flag                  bit(1) unal,
      2 type                  fixed bin(6) unsigned unal,
      2 packed                bit(1) unal,
      2 number_dims           fixed bin(4) unsigned unal,
      2 scale                 fixed bin(11) unal,
      2 precision             fixed bin(12) unsigned unal;

dcl 1 extended_arg_descriptor based aligned,
      2 flag                  bit(1) unal,
      2 type                  fixed bin(6) unsigned unal,
      2 packed                bit(1) unal,
      2 number_dims           fixed(4) unsigned unal,
      2 size                  bit(24) unal,
      2 dims                  (0 refer
                              (extended_arg_descriptor.number_dims)),
        3 low                 fixed bin(35),
        3 high                fixed bin(35),
        3 multiplier          fixed bin(35),
      2 real_type             fixed bin(18) unsigned unal,
      2 type_offset           fixed bin(18) unsigned unal;
```

The first four elements have the same meaning for all data where:

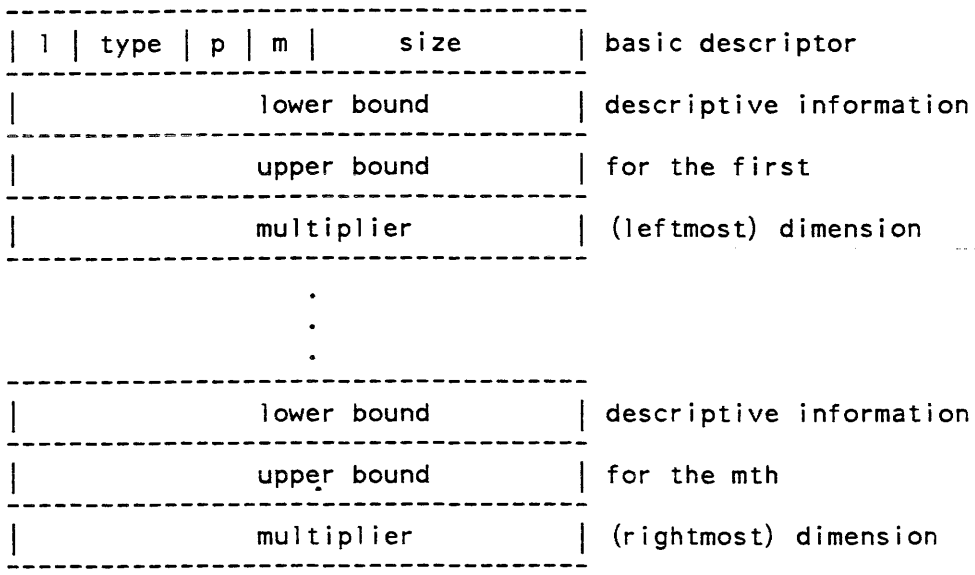
STRUCTURE ELEMENTS

flag
: always has the value "1"b and is used to tell this descriptor format from an earlier format. (Shown as 1 in the descriptor below.)

type
: is the data type according to the standard descriptor types (see Appendix D). Named constants for the descriptor types are declared in the std_descriptor_types.incl.pl1 include file. For extended descriptors (currently defined only for Pascal), type always has the value "58".

packed
: has the value "1"b if the data item is packed. (Shown as "p" in the typical descriptor below.)

number_dims
: is the number of dimensions in an array. (Shown as "m" in the descriptor below.) The array bounds and multipliers follow the basic descriptors in the following manner:



If the data is packed, the multipliers give the element separation in bits; otherwise, they give the element separation in words.

If the data is fixed-point, then:

scale
is a 2's complement, signed value.

precision
is the number of bits used to represent the data (if binary) or the number of digits (if decimal).

For all other data:

size
is the size (in bits, characters, or words) of string or area data, or the number of structure elements for structure data. In an argument descriptor for Algol68 array descriptor data, the size field is the number of dimensions of the array represented by the array decriptor datum. It is equal to the number_dims field of the second datum of the Algol68 array descriptor datum. In an argument descriptor for Algol68 union data, the size field is the number of words in the Algol68 union datum.

For extended descriptors:

low
is the lower bound of the dimension.

high
is the upper bound of the dimension.

multiplier
is the multiplier of the dimension.

real_type
is the data type for extended descriptors according to the standard descriptor types (see Appendix D).

type_offset
is the offset from the base of the symbol tree of the symbol node for the type, if any.

The descriptor of a structure is immediately followed by descriptors of each of its members. The example below shows a declaration (assuming that each element of C or D occupies one word) and its related descriptor.

```
dcl 1 S,
      2 A,
      2 B (5),
        3 C,
        3 D;
```

	basic descriptor of S
	basic descriptor of A
	basic descriptor of B
1	lower bound of B
5	upper bound of B
2	element separation of B
	basic descriptor of C
1	lower bound of C
5	upper bound of C
2	element separation of C
	basic descriptor of D
1	lower bound of D
5	upper bound of D
2	element separation of D

Members of dimensioned structures are arrays, and their descriptor contains copies of the bounds of the containing structure.

Parameter Descriptors

The parameter descriptors associated with an entry point have the same format as argument descriptors. The value 16777215 (77777777 octal) in the size field of an area, bit, or character parameter indicates an asterisk size. The value -34359738368 (400000000000 octal) in the lower bound, upper bound, or multiplier fields indicates asterisk array bounds.

APPENDIX I

DATA BASE DESCRIPTIONS

Listed below are descriptions of some Multics data bases presented in alphabetical order. Each description contains the name of the data base, discusses its purpose, and shows the correct usage.

Name

The "Name" heading shows the acceptable name by which the data base is referenced. The name is usually followed by a discussion of the purpose and function of the data base and the results that may be expected from referencing it.

Usage

This part of the data base description contains a declaration of the data base and its structure.

Name: sys_info

The sys_info data base is a per-system data base. It is accessible in all rings but can be modified only in ring 0. It contains many system parameters and constants. All references to it are made through externally defined variables.

STRUCTURE

```
dcl sys_info$clock_                    bit(3) aligned external static;
dcl 1 sys_info$ips_mask_data           aligned external static,
      2 count                          fixed binary,
      2 masks (sys_info$ips_mask_data.count),
        3 name                         char(32) aligned,
        3 mask                         bit(35)aligned;
dcl sys_info$page_size                 fixed binary(19) external static;
dcl sys_info$max_seg_size              fixed binary(19) external static;
dcl sys_info$default_stack_length      fixed binary(19) external static;
dcl sys_info$default_max_length        fixed binary(19) external static;
dcl sys_info$access_class_ceiling      bit(72) aligned external static;
dcl sys_info$time_correction_constant fixed binary(71) external static;
dcl sys_info$time_delta                fixed binary(35) external static;
dcl sys_info$maxlinks                  fixed binary external static;
dcl sys_info$time_of_bootload          fixed binary(71) external static;
dcl sys_info$time_zone                 char(3) aligned external static;
```

STRUCTURE ELEMENTS

clock_
is the port number of the system controller containing the clock.

ips_mask_data
is the array that specifies the number and mapping of interprocess signal (IPS) masks.

count
is the current number of valid IPS names.

name
is the name used to signal the IPS condition.

mask
is the IPS mask for the corresponding name. The mask has one bit on, and the rest of the bits are off.

page_size
is the page size in words.

max_seg_size
is the maximum segment size in words.

default_stack_length
is the default stack maximum size in words.

default_max_length
is the default maximum length of segments in words.

access_class_ceiling
is the maximum access class.

time_correction_constant
is the correction from Greenwich mean time (GMT) in microseconds.

time_delta
is the same as time_correction_constant, only in single precision.

maxlinks
is the maximum depth to which the system chases a link without finding a branch.

time_of_bootload
is the clock reading at the time of bootload.

time_zone
is the name of the time zone (e.g., EST).

NOTES

Variable factors in the user process can affect time as determined by time_correction_ constant, time_delta, and time_zone. The user is advised to avoid use of these values and see instead encode/decode_clock_value_, date_time_ and other similar subroutines in the Subroutines manual.

Name: whotab

The >scl>whotab segment is the public information data base for the system. All logged-in users, except those with the nolist attribute, have an entry in this table. These entries are listed by the who command. In addition, various system parameters of interest to all users are recorded in whotab. Many of these parameters are returned by the system_info_ subroutine and the system active function. Only the initializer process can modify the segment.

The structure of the whotab data base is as follows:

```
dcl 1 whotab                        based aligned
      2 mxusers                     fixed bin,
      2 n_users                     fixed bin,
      2 mxunits                     fixed bin,
      2 n_units                     fixed bin,
      2 timeup                      fixed bin (71),
      2 obsolete_sysid              char (8)
      2 nextsd                      fixed bin (71),
      2 until                       fixed bin (71),
      2 lastsd                      fixed bin (71),
      2 erfno                       char (8),
      2 obsolete_why                char (32),
      2 installation_id             char (32),
      2 obsolete_message            char (32),
      2 abs_event                   fixed bin (71),
      2 abs_procid                  bit (36),
      2 max_abs_users               fixed bin,
      2 abs_users                   fixed bin,
      2 n_daemons                   fixed bin
      2 request_channel             fixed bin (71),
      2 request_process_id          bit (36),
      2 shift                       fixed bin,
      2 next_shift_change_time      fixed bin (71),
      2 last_shift_change_time      fixed bin (71),
      2 fg_abs_users                fixed bin (17) unal,
      2 n_rate_structures           fixed bin (9) unsigned, unaligned,
      2 pad1                        bit (9) unaligned,
      2 pad (3)                     fixed bin,
      2 version                     fixed bin,
      2 header_size                 fixed bin,
      2 entry_size                  fixed bin,
      2 laste_adjust                fixed bin,
      2 laste                       fixed bin,
      2 freep                       fixed bin,
      2 header_extension_mbz1       fixed bin,
      2 n_abs (4)                   fixed fin,
      2 abs_qres (4)                fixed fin,
```

whotab

```
2 abs_cpu_limit (4)          fixed bin (35),
2 abs_control,
  3 mnbz                     bit (1) unaligned,
  3 abs_maxu_auto            bit (1) unaligned,
  3 abs_maxq_auto            bit (1) unaligned,
  3 abs_qres_auto            bit (1) unaligned,
  3 abs_cpu_limit_auto       bit (1) unaligned,
  3 queue_dropped (-1:4)     bit (1) unaligned,
  3 abs_up                   bit (1) unaligned,
  3 abs_stopped              bit (1) unaligned,
  3 control_pad              bit (24) unaligned,
2 installation_request_channel fixed bin (71),
2 installation_request_pid   bit (36),
2 sysid                      char (32),
2 header_extension_pad1 (7)  fixed bin,
2 header_extension_mbz2      fixed bin,
2 message                    char (124),
2 header_extension_mbz3      fixed bin,
2 why                        char (124),
2 e (1000),
  3 active                   fixed bin,
  3 person                   char (28),
  3 project                  char (28),
  3 anon                     fixed bin,
  3 padding                  fixed bin (71)
  3 timeon                   fixed bin (71),
  3 units                    fixed bin,
  3 stby                     fixed bin,
  3 idcode                   char (4),
  3 chain                    fixed bin,
  3 proc_type                fixed bin,
  3 group                    char (8),
  3 fg_abs                   bit (1) unaligned,
  3 disconnected             bit (1) unaligned,
  3 suspended                bit (1) unaligned,
  3 pad2                     bit (33) unaligned,
  3 cant_bump_until          fixed bin (71),
  3 process_authorization    bit (72);
```

STRUCTURE ELEMENTS

mxusers
: is the maximum number of users allowed on the system.

n_users
: is the current number of users.

mxunits
is the maximum number of load units allowed.

n_units
is the current load.

timeup
is the time the system was started.

obsolete_sysid
is obsolete; use the field sysid instead.

nextsd
is the time the system will be shutdown, if nonzero.

until
is the projected time of the next system start-up.

lastsd
is the time of last crash or shutdown.

erfno
is the error number of the last crash, if known.

obsolete_why
is obsolete; use why instead.

installation_id
is the name of the installation.

obsolete_message
is obsolete; use message instead.

abs_event
is the event channel for signalling absentee requests.

abs_procid
is the process identifier of the absentee user manager.

max_abs_users
is the current maximum number of absentee users.

abs_users
is the current number of absentee users.

n_daemons
is the number of daemons logged in via the message coordinator.

request_channel
is the event channel over which requests to the answering service should be sent.

request_processid
is the identifier of the process to which answering service requests should be sent.

shift
is the number of the current shift.

next_shift_change_time
is the time the current shift is scheduled to end.

last_shift_change_time
is the time the current shift started.

fg_abs_users
is the current number of foreground absentee users.

n_rate_structures
is the number of rate structures defined at the site.

pad1
is unused.

pad
is unused.

version
is the structure version (currently version 1).

header_size
is the length of the header (in words).

entry_size
is the length of an entry (in words).

laste_adjust
is used only by answering service programs. It gives the count of 32-word blocks in the header from header_extension_mbz1.

laste
is the index of the last entry in use.

freep
is the index of the first free entry chained through "chain."

header_extension_mbz1
is unused and is at offset 100 octal.

n_abs (4)
gives the number of processes from each background queue.

abs_qres (4)
gives the number of absentee positions reserved for each queue.

abs_cpu_limit (4)
gives the current absentee cpu limits.

abs_control
absentee control flags

mnbz
must not be zero.

abs_maxu_auto
is "1"b if automatic; "0"b if set manually.

abs_maxq_auto
is "1"b if automatic; "0"b if set manually.

abs_qres_auto
is "1"b if automatic; "0"b if set manually.

abs_cpu_limit_auto
is "1"b if automatic; "0"b if set manually.

abs_cpu_limit_auto
is "1"b if automatic; "0"b if set manually.

queue_dropped (-1:4)
is one if queue is dropped. Queue -1 is the foreground; 0-4 are respective background queue numbers.

abs_up
is "1"b if the absentee facility is running.

abs_stopped
is "1"b if the absentee facility is stopped. o.brp..argx control_pad is unused.

installation_request_channel
is the IPC channel for the install command.

installation_request_pid
is the installation process identifier.

sysid
is the current system name.

header_extension_pad1
is not used at present.

header_extension_mbz2
is unused and is at offset 140 octal.

message
is the message for all users.

header_extension_mbz3
is unused and is at offset 200 octal.

why
is the reason for the next shutdown.

User entry variables, with whotab.e(i):

active
is nonzero if this entry describes a logged-in user.

person
is the person name (Person_id).

project
is the project identifier (Project_id).

anon
indicates whether the user is an anonymous user:
1 yes
0 no

padding
is unused.

timeon
is the time of login.

units
is the number of load units for the user.

stby
indicates whether the user has secondary status:
1 yes
0 no

idcode
is the terminal identifier.

chain
is a chain for the free list.

proc_type
indicates the process type:
0 initializer
1 interactive
2 absentee
3 daemon

group
is the user's load-control group identifier.

fg_abs
is "1"b if this entry describes a foreground absentee user.

disconnected
is "1"b if the process is disconnected.

suspended
is "1"b if the process is suspended.

pad2
is unused.

cant_bump_until
is the time at which the user will (or did) become subject to preemption.

process_authorization
is the AIM authorization of the user's process.

APPENDIX J

STANDARD REQUEST TABLES AND STANDARD REQUESTS

This appendix contains descriptions of the standard request tables (of which there is currently one) and the standard requests provided as part of the subsystem utilities software.

STANDARD REQUEST TABLES

The following standard request tables are available for use in interactive subsystems using the subsystem utilities described in Section 4.

Name: ssu_request_tables_$standard_requests

This request table contains definitions of all standard requests available with the subsystem utilities.

USAGE

```
dcl ssu_request_tables_$standard_requests bit(36) aligned external;

... addr (ssu_request_tables_$standard_requests) ...
```

NOTES

The following standard requests are not listed by the summarize_requests request (i.e., their dont_summarize flag is set): summarize_requests, subsystem_name, subsystem_version, ready, ready_on, ready_off, and debug_mode.

The following standard requests are not listed by the list_requests request (i.e., their dont_list flag is set): subsystem_name, subsystem_version, ready, ready_on, ready_off, and debug_mode.

See "Subsystem Request Tables" in Section 4 for more information on the use of this table.

STANDARD REQUESTS

The remainder of this appendix contains command-like descriptions of the standard requests provided with the subsystem utilities, a list of which appears below by entry point and recommended request name.

Entry Point	Recommended Names
ssu_requests_$abbrev	abbrev, ab
ssu_requests_$answer	answer
ssu_requests_$debug_mode	debug_mode
ssu_requests_$do	do
ssu_requests_$exec_com	exec_com, ec
ssu_requests_$execute	execute, e
ssu_requests_$help	help
ssu_requests_$if	if
ssu_requests_$list_help	list_help, lh
ssu_requests_$list_requests	list_requests, lr
ssu_requests_$quit	quit, q
ssu_requests_$ready_off	ready_off, rdf
ssu_requests_$ready_on	ready_on, rdn
ssu_requests_$ready	ready, rdy
ssu_requests_$self_identify	.
ssu_requests_$subsystem_name	subsystem_name
ssu_requests_$subsystem_version	subsystem_version
ssu_requests_$summarize_requests	?

See "Using Standard Requests" in Section 4 for more information on the use of these requests.

Recommended Names: abbrev, ab

SYNTAX AS A REQUEST

```
ab {-control_args}
```

SYNTAX AS AN ACTIVE FUNCTION

```
[ab]
```

FUNCTION

controls abbreviation processing within the subsystem. As an active request, returns "true" if abbreviation expansion of request lines is currently enabled within the subsystem and "false" otherwise.

CONTROL ARGUMENTS

Control arguments may not be used with the active request.

-off
specifies that abbreviations are not to be expanded.

-on
specifies that abbreviations should be expanded (default).

-profile PATH
specifies that the segment named by PATH is to be used as the profile segment; the suffix ".profile" is added to PATH if not present. The segment named by PATH must exist.

NOTES

Most subsystems which support abbreviation processing provide command line control arguments (-abbrev, -no_abbrev, -profile) to specify the initial state of abbreviation processing within the subsystem. For example, a Multics abbreviation could be defined to invoke the read_mail subsystem with a default profile as follows:

```
.ab rdm do "read_mail -abbrev [hd]>mail_system &rfl"
```

If invoked with no arguments, this request will enable abbreviation processing within the subsystem using the profile that was last used in this subsystem invocation. If abbreviation processing was not previously enabled, the profile in use at Multics command level is used; this profile is normally `[home_dir]>Person_id.profile`.

See *Multics Commands and Active Functions,* Order No. AG92, for a description of abbreviation processing.

Recommended Names: answer

SYNTAX AS A REQUEST

```
answer STR {-control_args} request_line
```

FUNCTION

provides preset answers to questions asked by another request.

ARGUMENTS

STR
is the desired answer to any question. If the answer is more than one word, it must be enclosed in quotes. If STR is -query, the question is passed on to the user. The -query control argument is the only one that can be used in place of STR.

request_line
is any subsystem request line. It can contain any number of separate arguments (i.e., have spaces within it) and need not be enclosed in quotes.

CONTROL ARGUMENTS

-brief, -bf
suppresses printing (on the user's terminal) of both the question and the answer.

-call STR
evaluates the active string STR to obtain the next answer in a sequence. The active string is constructed from subsystem active requests and Multics active strings (using the subsystem's execute active request). The outermost level of brackets must be omitted (i.e., "list_meetings -changes"), and the entire string must be enclosed in quotes if it contains request processor special characters. The return value "true" is translated to "yes", and "false" to "no". All other return values are passed as is.

-match STR
answers only questions whose text matches STR. If STR is surrounded by slashes (/), it is interpreted as a qedx regular expression. Otherwise, answers tests whether STR is literally contained in the text of the question. Multiple occurrences of -match and -exclude are allowed (See Notes below). They apply to the entire request line.

-exclude STR, -ex STR
passes on, to the user or other handler, questions whose text matches STR. If STR is surrounded by slashes (/), it is interpreted as a qedx regular expression. Otherwise, answer tests whether STR is literally contained in the text of the question. Multiple occurrences of -match and -exclude are allowed (see Notes below). They apply to the entire request line.

-query
skips the next answer in sequence, passing the question on to the user. The answer is read from the user_i/o I/O switch.

-then STR
supplies the next answer in a sequence.

-times N
gives the previous answer (STR, -then STR or -query) N times only (where N is an integer).

NOTES

Answer provides preset repsonses to questions by establishing an on unit for the condition command_question., and then executing the designated request line. If any request in the request line calls the command_query_ subroutines (described in *Multics Subroutines and I/O Modules,* Order No. AG93, to ask a question, the on unit is invoked to supply the answer. The on unit is reverted when the answer request returns to subsystem request level.

If a question is asked that requires a yes or no answer, and the preset answer is neither "yes" nor "no", the on unit is ignored and the user will be queried.

The last answer specified is issued as many times as necesary, unless followed by the -times N control argument.

The -match and -exclude control arguments are applied in the order specified. Each -match causes a given question to be answered if it matches STR, each -exclude causes it to be passed on if it matches STR. A question that has been excluded by -exclude is reconsidered if it matches a -match later in the request line. For example, the request line:

```
answer  yes -match /fortran/ -exclude  /fortran_io/
        -match /^fortran_io/
```

answers questions containing the string "fortran" except that it does not answer questions containing "fortran_io", except that it does answer questions beginning with "fortran_io".

Recommended Names: debug_mode

SYNTAX AS A REQUEST

```
debug_mode -control_arg
```

FUNCTION

enables or disables debugging mode for the current subsystem.

CONTROL ARGUMENTS

-on
    enables debugging mode for this subsystem.

-off
    disables debugging mode for this subsystem.

NOTES

This request is intended for use only by subsystem implementors.

See "Subsystem Debugging Facilities" in Section 4 for further information.

Recommended Names: do

SYNTAX AS A REQUEST

```
do request_string {args}
or
do -control_args
```

SYNTAX AS AN ACTIVE FUNCTION

```
[do "request_string" args]
```

FUNCTION

expands a request line by substituting the supplied arguments into the line before execution. As an active request, returns the expanded request_string rather than execute it.

ARGUMENTS

request_string
is a request line in quotes.

args
are character string arguments that replace parameters in request_string.

CONTROL ARGUMENTS

These control arguments set the mode of operation of the do request:

-long, -lg
the expanded request line is printed before execution.

-brief, -bf
the expanded request line is not printed before execution (default).

-nogo
the expanded request line is not passed on for execution.

-go
the expanded request line is passed on for execution (default).

-absentee
an any_other handler is established which catches all conditions and aborts execution of the request without aborting the process.

-interactive
the any_other handler is not established (default).

PARAMETERS

Any sequence beginning with & in the request line is expanded by the do request using the arguments given on the request line.

&I
is replaced by argI; I must be a digit from 1 to 9.

&(I)
is also replaced by argI; I can be any value, however.

&qI
is replaced by argI with any quotes in argI doubled; I must be a digit from 1 to 9.

&q(I)
is also replaced by argI with any quotes doubled; I can be any value.

&rI
is replaced by argI surrounded by a level quotes with any contained quotes doubled. I must be a digit from 1 to 9.

&r(I)
is also replaced by a requoted argI; I can be any value.

&fI
is replaced by all the arguments starting with argI; I must be a digit from 1 to 9.

&f(I)
is also replaced by all the arguments starting with argI; I can be any value.

&qfI
is replaced by all the arguments starting with argI with any quotes doubled; I must be a digit from 1 to 9.

&qf(I)
is also replaced by all the arguments starting with argI with quotes doubled; I can be any value.

&rI
is replaced by all the arguments starting with argI. Each argument is placed in a level of quotes with contained quotes doubled; I must be a digit from 1 to 9.

&rf(I)
is also replaced by all the arguments starting with argI, requoted; I can be any value.

&&
is replaced by an ampersand.

&!
is replaced by a 15-character unique string. The string used is the same everywhere &! appears in a request line.

&n
is replaced by the actual number of arguments supplied.

&f&n
is replaced by the last argument supplied.

Recommended Names: exec_com, ec

SYNTAX AS A REQUEST

```
ec ec_path {ec_args}
```

SYNTAX AS AN ACTIVE FUNCTION

```
[ec ec_path {ec_args}]
```

FUNCTION

executes a program written in the exec_com language which is used to pass request lines to the subsystem and to pass input lines to requests which read input. As an active function, the exec_com program specifies a return value of the exec_com request by use of the &return statement.

ARGUMENTS

ec_paths
is the pathname of an exec_com program. The suffix, which is normally the name of the subsystem, is assumed if not specified.

ec_args
are optional arguments to the exec_com program and are substituted for parameter references in the program such as &1.

NOTES

Subsystems may define a search list to be used to find the exec_com program. If this is the case, the search list is used if ec_path does not contain a "<" or ">" character; if the ec_path does contain either a "<" or ">", it is assumed to be a relative pathname.

For a description of the exec_com language (both version 1 and version 2), type:

```
.. help v1ec v2ec
```

When evaluating a subsystem exec_com program, subsystem active requests are used rather than Multics active functions when evaluating the &[...] construct and the active string in an &if statement. The subsystem's execute active request may be used to evaluate Multics active strings within the exec_com.

Recommended Names: execute, e

SYNTAX AS A REQUEST

```
e LINE
```

SYNTAX AS AN ACTIVE FUNCTION

```
[e LINE]
```

FUNCTION

executes the supplied line as a Multics command line. An as active request, evaluates a Multics active string and returns the result to the subsystem request processor.

ARGUMENTS

LINE
is the Multics command line to be executed or the Multics active string to be evaluated. It need not be enclosed in quotes.

NOTES

The recommended method to execute a Multics command line from within a subsystem is the ".." escape sequence. The execute request is intended as a means of passing information from the subsystem to the Multics command processor.

All (), [], and "'s in the given line are processed by the subsystem request processor and not the Multics command processor. This fact permits the passing of the values of subsystem active requests to Multics commands when using the execute request or, when using the execute active request, to Multics active functions for further manipulation before returning the values to the subsystem request processor for use within a request line.

EXAMPLES

The read_mail request line:

```
e mbla [mailbox]
```

can be used to list the ACL of the mailbox being read by the current invocation of read_mail.

The read_mail request line:

```
write all [e strip_entry [mailbox]]
```

may be used to write the ASCII representation of all messages in the mailbox into a segment in the working directory whose entry name is the same as that of the mailbox with "mbx" suffix changed to "mail".

Recommended Names: help

SYNTAX AS A REQUEST

```
help {topics} {-control_args}
```

FUNCTION

prints information about various subsystem topics including detailed descriptions of most subsystem requests.

ARGUMENTS

topics
: are the topics on which information is to be printed. The topics available within a subsystem can be determined by using the list_help request if available.

CONTROL ARGUMENTS

The most useful control arguments are:

-title
prints section titles and section line counts, then asks if the user wants to see the first paragraph of information.

-brief, -bf
prints only a summary of a request or active request, including the Syntax section, list of arguments, control arguments, etc.

-section STRs, -scn STRs
begins printing at the section whose title contains all the strings STRs. By default, printing begins at the top of the information.

-search, STRs, -srh STRs
begins printing with the paragraph containing all the strings STRs. By default, printing begins at the top of the information.

RESPONSES

The most useful responses which can be given to questions asked by the help request are:

yes, y
prints the next paragraph of information on this topic.

no, n
stops printing information for this topic and proceeds to the next topic if any.

quit, q
stops printing information for this topic and returns to the subsystem's request level.

`rest {-scn}, r {-scn}`
prints remaining information of this topic without intervening questions. If -section or -scn is given, help prints only the rest of the current section without questions and then asks if the user wants to see the next section.

`title {-top}`
lists titles and line counts of the sections which follow; if -top or -t is given, help lists all section titles. help then repeats the previous question after titles are printed.

`section {STRs} {-top}, scn {STRs} {-top}`
skips to the next section whose title contains all the strings STRs. If -top or -t is given, title searching starts at the top of the information. If STRs are omitted, help uses the STRs from the previous section response or the -section control argument.

`search {STRs} {-top}, srh {STR} {-top}`

skips to the next paragraph containing all the strings STRs. If -top or -t is given, searching starts at top of the information. If STRs are omitted, help uses the STRs from the previous search response or the -search control argument.

`skip {-scn} {-seen}, s {-scn} {-seen}`

skips to the next paragraph. If -section or -scn is given, skips all paragraphs of the current section. If -seen is given, skips to the next paragraph which the user has not seen. Only one control argument is allowed in each skip response.

?

prints the list of responses allowed to help queries.

prints "help" to identify the current interactive environment.

.. command_line

treats the remainder of the response as a Multics command line.

NOTES

If given no topic names, the help request will explain what requests are available in the subsystem to obtain information on the subsystem.

For a complete description of the control arguments and responses accepted by this request, type:

.. help help

Recommended Names: if

SYNTAX AS A REQUEST

`if EXPR -then LINE1 {-else LINE2}`

SYNTAX AS AN ACTIVE FUNCTION

`if EXPR -then STR1 {-else STR2}`

FUNCTION

conditionally executes one of two request lines depending on the value of an active string. As an active request, returns one of two character strings to the subsystem request processor depending on the value of an active string.

ARGUMENTS

EXPR
is the active string which must evaluate to either "true" or "false". The active string is constructed from subsystem active requests and Multics active strings (using the subsystem's execute active request).

LINE1
is the subsystem request line to execute if EXPR evaluates to "true". If the request line contains any request processor characters, it must be enclosed in quotes.

STR1
is returned as the value of the if active request if the EXPR evaluates to "true".

LINE2
is the subsystem request line to execute if EXPR evaluates to "false". If omitted and EXPR is "false", no additional request line is executed. If the request line contains any request processor characters, it must be enclosed in quotes.

STR2
is returned as the value of the if active request if the EXPR evaluates to "false". If omitted and the EXPR is "false", a null string is returned.

Recommended Names: list_help, lh

SYNTAX AS A REQUEST

```
lh {topics}
```

FUNCTION

displays the names of all subsystem info segments pertaining to a given set of topics.

ARGUMENTS

topics
specifies the topics of interest. Any subsystem info segment which contains one of these topics as a substring is listed.

NOTES

If no topics are given, all info segments available for the subsystem are listed.

When matching topics with info segment names, an info segment name is considered to match a topic only if that topic is at the beginning or end of a word within the segment name. Words in info segment names are bounded by the beginning and end of the segment name and by the characters period (.), hyphen (-), underscore (_), and dollar sign ($). The ".info" suffix is not considered when matching topics.

EXAMPLES

The request line:

```
list_help list
```

will match info segments named list, list_users, and forum_list but will not match an info segment named prelisting.

Recommended Names: list_requests, lr

SYNTAX AS A REQUEST

```
lr {STRs} {-control_args}
```

FUNCTION

prints a brief description of selected subsystem requests.

ARGUMENTS

STRs
: specifies the requests to be listed. Any request with a name containing one of these strings is listed unless -exact is used in which case the request name must exactly match one of these strings.

CONTROL ARGUMENTS

-all, -a
: includes undocumented and unimplemented requests in the list of requests eligible for matching the STR arguments.

-exact
: lists only those requests one of whose names exactly match one of the STR arguments.

NOTES

If no STRs are given, all requests are listed.

When matching STRs with request names, a request name is considered to match a STR only if that STR is at the beginning or end of a word within the request name. Words in request names are bounded by the beginning and end of the request name and by the characters period (.), hyphen (-), underscore (_), and dollar sign ($).

EXAMPLES

The request line:

```
list_requests list
```

will match requests named list, list_users, and forum_list but will not match a request named prelisting.

Recommended Names: quit, q

SYNTAX AS A REQUEST

```
q
```

FUNCTION

exits the subsystem and returns to Multics command level.

Recommended Names: ready_off, rdf

SYNTAX AS A REQUEST

```
rdf
```

FUNCTION

turns off printing of ready messages after execution of each subsystem request line.

Recommended Names: ready_on, rdn

SYNTAX AS A REQUEST

```
rdn
```

FUNCTION

turns on printing of ready messages after execution of each subsystem request line.

Recommended Names: ready, rdy

SYNTAX AS A REQUEST

```
rdy
```

FUNCTION

prints a ready message.

NOTES

The Multics general_ready command may be used to change the format of the ready message printed by this request and also after execution of request lines if the ready_on request is used. Type:

.. help general_ready

for more information on the available formats.

The default ready message gives the time of day and the amount of CPU time and page faults used since the last ready message was typed.

Recommended Names: self_identify

SYNTAX AS A REQUEST

FUNCTION

prints the name and version of the subsystem, whether abbreviation processing is enabled in the subsystem, and the level of invocation of this instance of the subsystem if more than one invocation of the subsystem is active.

EXAMPLES

debug_tpsa 6.1a (abbrev) (level 3)

Recommended Names: subsystem_name

SYNTAX AS A REQUEST

```
subsystem_name
```

SYNTAX AS AN ACTIVE FUNCTION

```
[subsystem_name]
```

FUNCTION

prints the name of the subsystem; as an active request, returns the name of the subsystem.

Recommended Names: subsystem_version

SYNTAX AS A REQUEST

```
subsystem_version
```

SYNTAX AS AN ACTIVE FUNCTION

```
[subsystem_version]
```

FUNCTION

prints the version number of the subsystem; as an active request, returns the version number of the subsystem.

Recommended Names: ?

SYNTAX AS A REQUEST

?

FUNCTION

prints a list of requests available in this subsystem.

NOTES

There may be other requests available in this subsystem which are not listed by this request. Such requests may be listed using the request:

list_requests -all

INDEX

C

E

M

N

P

Q

R

T

U

V

W

Z

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS
PROGRAMMER'S REFERENCE MANUAL
ADDENDUM A

ORDER NO. AG91-04A

DATED FEBRUARY 1987

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

PLEASE FILL IN COMPLETE ADDRESS BELOW.

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

CUT ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U.S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS PROGRAMMER'S REFERENCE MANUAL

ORDER NO. AG91-04

DATED FEBRUARY 1985

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho, Chiyoda-ku, Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

42843, 9C585, Printed in U.S.A.

AG91-04