Honeywell

MULTICS PROGRAMMERS' MANUAL
REFERENCE GUIDE

SERIES 60 (LEVEL 68)

SOFTWARE

Honeywell

MULTICS PROGRAMMERS' MANUAL REFERENCE GUIDE

SERIES 60 (LEVEL 68)

SUBJECT:

Reference Guide to the Overall Mechanics, Conventions, and Usage of the Multics System.

SPECIAL INSTRUCTIONS:

This manual is one of four manuals that constitute the Multics Programmers' Manual (MPM).

Reference Guide	Order No. AG91
Commands and Active Functions	Order No. AG92
Subroutines	Order No. AG93
Subsystem Writers' Guide	Order No. AK92

This manual supersedes AG91, Rev. 0. The manual has been extensively revised; therefore, marginal change indicators have not been included in this edition.

SOFTWARE SUPPORTED:

Multics Software Release 3.0

DATE:

December 1975

ORDER NUMBER:

AG91, Rev. 1

PREFACE

Primary reference for user and subsystem programming on the Multics system is contained in four manuals. The manuals are collectively referred to as the Multics Programmers' Manual (MPM). Throughout this manual, references are frequently made to the MPM. For convenience, these references will be as follows:

Document	Referred To In Text As
Reference Guide (Order No. AG91)	MPM Reference Guide
Commands and Active Functions (Order No. AG92)	MPM Commands
Subroutines (Order No. AG93)	MPM Subroutines
Subsystem Writers' Guide (Order No. AK92)	MPM Subsystem Writers' Guide

The MPM Reference Guide contains general information about the Multics command and programming environments. It also defines items used throughout the rest of the MPM. And, in addition, describes such subjects as the command language, the storage system, and the input/output system.

The MPM Commands is organized into four sections. Section I contains a list of the Multics command repertoire, arranged functionally. It also contains a discussion on constructing and interpreting names. Section II describes the active functions. Section III contains descriptions of standard Multics commands, including the calling sequence and usage of each command. Section IV describes the requests used to gain access to the system.

The MPM Subroutines is organized into three sections. Section I contains a list of the subroutine repertoire, arranged functionally. Section II contains descriptions of the standard Multics subroutines, including the declare statement, the calling sequence, and usage of each. Section III contains the descriptions of the I/O modules.

The MPM Subsystem Writers' Guide is a reference of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules that allow the user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the majority of users.

© 1975, Honeywell Information Systems Inc.

Examples of specialized subsystems for which construction would require reference to the MPM Subsystem Writers' Guide are:

- A subsystem that precisely imitates the command environment of some system other than Multics.
- A subsystem intended to enforce restrictions on the services available to a set of users (e.g., an APL-only subsystem for use in an academic class).
- A subsystem that protects some kind of information in a way not easily expressible with ordinary access control lists (e.g., a proprietary linear programming system, or an administrative data base system that permits access only to program-defined, aggregated information such as averages and correlations).

Several cross-reference facilities help locate information:

- Each manual has a table of contents that identifies the material (either the name of the section and subsection or an alphabetically ordered list of command and subroutine names) by page number.
- Each manual contains an index that lists items by name and page number.

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

ILLUSTRATIONS

TABLES

SECTION I

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 refers the reader to either a section of this manual or to another manual is supplied with most of the terms defined.

HISTORY OF MULTICS

The Multics system is a general purpose computer system developed by Honeywell Information Systems. Introduced to commercial markets in January 1973, Multics was then the result of more than 7 years of joint research on the part of the development principals. Continued development and system enhancement is planned to extend well into the future. Multics was implemented initially on the Honeywell Model 645 computer system, an enhanced relative of the Honeywell Model 635 computer system. A transition was later made to the more advanced Honeywell Model 6180 computer system when it became available, and now has been made to Honeywell's top-of-the-line Series 60/Level 68 system.

SYSTEM CONCEPTS

Multics is a unique combination of hardware, software, communications capabilities, and supervisory techniques. The system hardware, particularly the central processor, is specifically designed for the Multics operating system. The system provides capabilities that have long been sought by research, government, academic, and network-oriented computer users -- those sophisticated 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.

An 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, in more detail, 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 256K 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. Each directory segment also lists the attributes including length, secondary storage address, date and time of creation, and access restrictions for each segment cataloged. 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 "waited" 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 how 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 among users' processes. A user's process executes programs sequentially. A user's process also has an address space. Each process can directly reference only those segments that have been assigned a segment number for that process by supervisor calls.

The system creates a process for each user at login time. (For information on logging in and out on the system, see "How to Access the Multics System" in the Multics Users' Guide, Order No. AL40.) Within the constraints imposed by the supervisor, a user may customize his process as desired: the commands, command processor and environment, and translator provided by the system can all be replaced in a user's process by his own code. A project administrator also has the option of enforcing the use of a given process environment upon users in his project.

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). All processes coexist in the system, and cannot maliciously or accidentally interfere with each other.

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. No per-user "copies" or "images" of segments exist. 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 or malicious 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. The access permission checking is still required, 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 II, "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. (See the System Administrator's Manual, Order No. AK50 and the Project Administrator's Manual, Order No. AK51.)

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. All of the interfaces available to system-provided commands and subsystems are available to the user and are documented in the MPM Subsystem Writers' Guide.

Programming Environment

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

Programming Languages

The Multics system includes several language processors. A program written in one of the Multics languages can call programs written in another language by merely following appropriate calling conventions. The Multics compilers optionally generate a symbol table that permits a user to check out his program at the original source level with the aid of the debug, probe, and trace commands. (For detailed information about programming on Multics in the various programming languages, see the language manual and users' guide for the languages in question.)

PL/I

The PL/I compiler for Multics offers a full selection of language facilties and access to the advanced features of Multics. PL/I is the recommended programming language for Multics users. Multics PL/I conforms to the draft American National Standards Institute standard for the language.

FORTRAN

The Multics FORTRAN compiler conforms to the current American National Standards Institute (ANSI) for FORTRAN. Multics FORTRAN is a superset of the ANSI standard FORTRAN and is source-language-compatible with Series 60/Level 66 FORTRAN.

BASIC

The Multics BASIC is compatible with the Dartmouth Version 6 BASIC and contains all the functional capabilities of the Series 60/Level 66 BASIC compiler.

COBOL

This compiler is a subset of the ANSI standard COBOL and is source-language-compatible with Series 60/Level 66, COBOL-74.

APL

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

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.

runoff

The runoff command, used for manuscript formatting, also has programmable requests that make it a minor programming language.

Support Facilities and Tools

Stable and reliable software components within the Multics operating system provide numerous utility and support functions. Foremost among these are the Multics text editors, edm and qedx.

The debug, probe, 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.

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 Section I of the MPM Commands, Section I of the MPM Subroutines, and Section V of the MPM Subsystem Writers' Guide.

REMOTE ACCESS

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 the 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 "governors" that limit the amount of system resources that 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, approximately 95% of the Multics operating system is written in PL/I, which makes further flexibility and easy modification possible.

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 an incremental backup system that dumps 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 the backup tapes and its reinclusion in the online storage system. Finally, there 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.

GLOSSARY OF MULTICS TERMS

absentee

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

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 VI.)

access control

The mechanism for determining who may reference or modify segments (files) and directories. (See "Discretionary Access Control" in Section VI.)

access control list (ACL)

A set of access identifiers specifying who may 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 VI.)

access identifier (access_id, access_name)

A character string representing a user or class of users. It consists of three fields: Person_id.Project_id.tag. (See "Access Identifier" in Section VI.)

access isolation mechanism (AIM)

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

access modes

A way to identify the kinds of access that may 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 VI.)

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 III.)

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 MPM Commands.)

alternate name(s)

A segment, directory, multisegment file, or link may 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. 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.)

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 MPM Commands.)

attach

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

authorization

An access isolation mechanism (AIM) attribute that denotes the range of information a process can access. An authorization is associated with a process for its lifetime. (See "Nondiscretionary Access Control" in Section VI.)

backup

The 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 VIII, "Backup and Retrieval.")

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 II.)

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 MPM Commands.)

branch

An item cataloged in a directory: segment, multisegment file or another directory but not a link. (See entry below.)

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. (See "Canonicalization" in Section IV.)

"carriage return"

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

closed subsystem

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

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 can be used as a command. (See "Command Language" in Section III.)

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 when he logs in, 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 III.)

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 III.)

component (of an archive)

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

component (of an entryname)

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

con_msgs

See Person_id.con_msgs below.

control argument

An argument to a command that specifies what the user wants done, or what information he is interested in. 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. A list of the abbreviations of the most frequently used control arguments is found in Appendix A of the MPM Commands. (System commands are described in the MPM Commands and in the MPM Subsystem Writers' Guide.)

crash

An unplanned termination of system availability caused by problems in hardware and/or software.

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 segments, multisegment files, links and other subordinate directories. The directory contains information about the attributes of these entries and information about the physical device on which the data is stored. (See Section II, "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 (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 IV.)

directory hierarchy

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

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 dprint and dpunch commands in the MPM Commands.)

dump

See backup above.

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 IV.)

entry

An item cataloged in a directory: segment, link, multisegment file, or another directory. (See branch and file in this glossary and "Entrynames" in Section III.)

entry point

An address in an object segment referenced 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 III.)

entryname

A name given to an item contained in a directory. 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 III.)

equal convention

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

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 MPM Commands.)

extended access

An additional field of access modes used with message segments to further restrict operations on a message segment. (See "Access Modes" in Section VI.)

fault

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

file

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

frame

See main memory frame 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 VI.)

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.

home directory

See directory (home) above.

impure procedure

A procedure that modifies itself.

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_segments (>doc>info). The info segments that are peculiar to an installation are kept in >doc>iml_info_segments. (See the help command in the MPM Commands.)

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 VI.)

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 III.)

Installation Maintained Library (IML)

The library of programs maintained by the particular installation. It is kept in the directory >system_library_auth_maint (>am). (See Section II, "Storage System.")

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 module of the system are stored. (See Section II, "Storage System.")

limited service system

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

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.

mailbox

See Person_id.mbx.

memory units

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

making a segment known

Specifying its pathname 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 IV.)

main memory frame

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

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 Programmers' Manual (MPM)

The primary reference manual for Multics. (See the preface of this document.)

Multics card code (MCC)

A code for punched card input and output. It is essentially the IBM standard EBCDIC card code. This is not 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 262,144 words. May only be manipulated by certain programs. (See "Multisegment Files" in Section II.)

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 be searched and linked to by the dynamic linking mechanism. (See "Creating an Object Segment" in Section IV.)

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 I.)

password

A character string supplied by a user and known only to him and the software 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. (See Section I, "How to Access the Multics System", in the Multics Users' Guide, Order No. AL40.)

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 III.)

pathname (absolute)

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

pathname (relative)

A pathname that uniquely names a segment relative to the working directory. (See "Pathnames" in Section III.)

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 I, "How to Access the Multics System," in the Multics Users' Guide, Order No. AL40.)

Person_id.con_msgs

A segment in the user's home directory used to hold messages sent by the send_message command. (See the send_message and accept_message commands in the MPM Commands.)

Person_id.mbx

A message segment used to convey, between processes, arbitrary text (a letter) intended to be read by a user. (See the mail command in the MPM Commands.)

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 MPM Commands.)

process

A program or group of programs in execution: an address space and an execution point. Each logged-in user has his own process. (See "Process" in Section I.)

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.

process overseer

The first procedure called in a process. It sets up the environment, 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)

A compiled project master file.

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 ATTN, BRK, INTERRUPT, etc. key on a terminal. This condition normally causes the printing of QUIT followed by establishment of a new command level. (See "System Conditions" in Section VII.)

quote

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

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 own ready message. The standard system ready message tells the time of day and the number of CPU seconds, memory units, and page faults since the last ready message plus the current listener level (if greater than 1).

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, particular names may be associated with it in that process. This is called initiation. Thereafter, a symbolic reference to this reference name is directed to the associated segment. Reference names need not be the same as any of the segment's entrynames. (See "Reference Names" in Section III.)

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 VIII, "Backup and Retrieval.")

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 II, "Storage System.")

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 "Intraprocess Access Control" in Section VI.)

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 "Intraprocess Access Control" in Section VI.)

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. This is to be distinguished from addressing a segment by its pathname, which explicitly specifies the directory containing the segment. (See "Search Rules" in Section II.)

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 null. (See "Segments" in Section I.)

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) 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, >system_library_unbundled, and >system_library_1. (See Section II, "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 IV.)

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 III.)

start_up.ec

An exec_com segment that is invoked automatically when the user logs in. It is often used to execute commands such as mail, abbrev, and accept_messages.

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 VIII.)

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 (components are separated by a period (.)) that usually specifies the type of segment. For example, pl1, con_msgs, and list. A segment without a suffix is usually an object segment or data segment. (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 V, "Input and Output Facilities.")

SysDaemon

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_control_dir (sc1, 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., 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 III.)

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 IV.)

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 III.)

User_id
Used to refer to a Person_id.Project_id pair. (See access identifier above.)

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).

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 unit of information that on Multics is 36 bits.

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

SECTION II

MULTICS STORAGE SYSTEM

The basic unit of storage in the Multics storage system is a segment. Segments form a tree-structured data base that is organized by a hierarchy of special segments called directories. As shown in Figure 2-1, any segment (directory or nondirectory) in the tree 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. The immediately superior directory also referred to as the containing directory.

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). Using Figure 2-1 as a reference, the absolute pathname for the segment named "chess" would be:

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

The syntax of entrynames and pathnames is given in detail in "Entrynames" and "Pathnames" in Section III. The following discussion gives the general contents of a directory and describes the system directories that form the basis of the storage hierarchy.

DIRECTORY CONTENTS

A directory contains a series of entries, each of which is used to locate a target segment. The target segment can be a procedure or data segment or another directory. An entry is composed of the segment's name (entryname) and may be one of two kinds -- a branch or a link. A branch contains a full set of attributes describing properties of the segment such as its physical location, length, access rights, and so on. A link entry contains a pathname leading to the target segment through an entry in some other directory. A particular link may point directly to the entry of interest, to another link entry, or, since no checking is done, to a nonexistent entry.

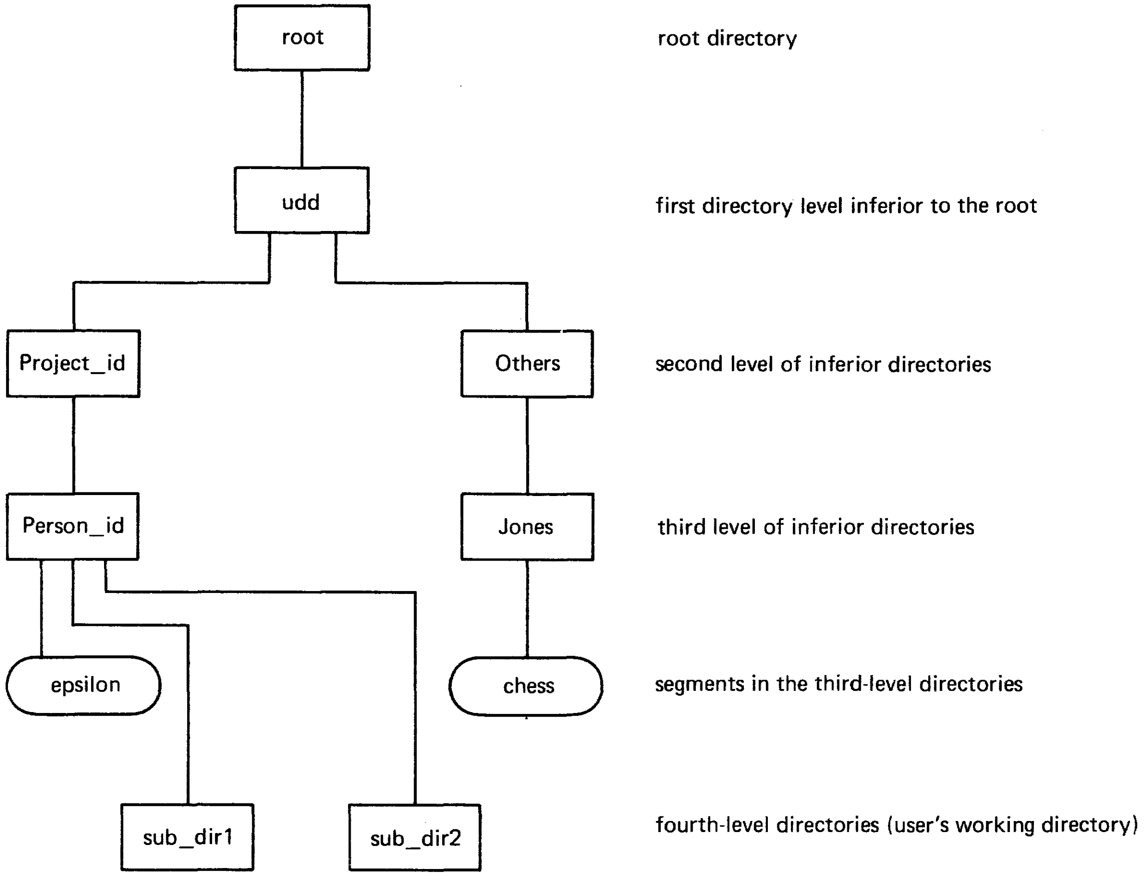

root
root directory
udd
first directory level inferior to the root
Project_id
Others
second level of inferior directories
Person_id
Jones
third level of inferior directories
epsilon
chess
segments in the third-level directories
sub_dir1
sub_dir2
fourth-level directories (user's working directory)

Figure 2-1. Storage System Hierarchy

Entry Attributes

A branch-type entry serves both to identify and characterize the target segment. Attributes are entered when the segment is created and, in most cases, can be modified later either explicitly or implicitly. As shown below, the attributes maintained in a branch entry differ for the three possible branch entry types: segment, directory, or multisegment file.

Explicit modification of an attribute is achieved using a standard storage system subroutine. Implicit modification is automatic and occurs as a result of some change to the target segment. For example, when data is written into an existing segment, the date-time-modified attribute is changed accordingly.

The types of attributes maintained in a directory and the types of entry to which they are applicable are described below. Multisegment files are treated somewhat differently and are described in "Multisegment Files" (following this attribute list).

access class (segments, directories)

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 "Nondiscretionary Access Control" in Section VI.

access control list (segments, directories)

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 "Discretionary Access Control" in Section VI for a complete discussion of access control.

author (segments, directories, links)

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

bit count (segments)

The bit count attribute gives the length (in bits) of the segment. 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 (segments)

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.

copy switch (segments)

The copy switch attribute is a mechanism that permits simultaneous executions of an impure procedure by more than one process. If the copy switch is "on", each process that refers to the entry is given a copy of the segment rather than the segment itself. The copy switch may also be used to obtain a copy of a data segment. Since a new copy is also generated each time an external reference invokes the segment by a different name, it is recommended that only one name be associated with a segment whose copy switch is on. This attribute can be explicitly modified.

current length (segments, directories)

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.

date time dumped (segments, directories, links)

This attribute records the time at which a backup copy of the entry was last made by the Multics backup procedures. The date-time-dumped attribute is automatically modified by these procedures.

date time entry modified (segments, directories, links)

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

date time modified (segments, directories)

This attribute records the time at which an entry was last modified. It is implicitly updated as a result of the modification. (This value is approximate and is normally within a few minutes of the time the entry was modified.)

date time salvaged (directories)

This attribute records when the directory was last salvaged. The term salvage refers to the rebuilding of a directory, undertaken either as a corrective or housekeeping measure. This attribute is implicitly modified as a result of the salvaging process.

date time used (segments, directories, links)

This attribute records the last time the target entry was referenced. The date-time-used attribute is implicitly updated when the entry is used. (This value is approximate and is normally within a few minutes of the last time the entry was modified.)

initial access control lists (directories)

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 "Initial ACLs" in Section VI 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.

maximum length (segments)

The maximum length attribute sets a limit on the size a segment can attain. Maximum length is accurate to units of 16 words and can be explicitly modified. The maximum value in words is 256K (K = 1024).

multisegment file indicator (directories)

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 in the file (that is, the entryname of the last segment in the file plus one). The multisegment file indicator is implicitly modified by multisegment file primitives when the length of the file changes. It can also be explicitly modified.

names (segments, directories, links)

The names attribute is one or more character strings that identify the entry. Each entry can have many names. The first name returned to the storage system is called the primary name. Entrynames can be explicitly modified.

quota (directories)

The quota attribute gives the maximum number of storage records permitted to segments and directories inferior to the target directory. This value excludes subtrees that have their own quotas. The quota attribute can be explicitly modified by any process that has modify access in the directory at the validation level. The modification is done by moving partial amounts to or from the superior directory. The total quota in the hierarchy is constant.

records used (segments, directories)

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.

ring brackets (segments, directories)

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

safety switch (segments, directories)

The safety switch attribute is used to protect an entry from deletion. If the safety switch is "on", the user is asked if the target entry should be deleted before the deletion is performed. This attribute can be explicitly modified.

secondary storage device identifier (segments, directories)

This attribute identifies the type of device on which the target entry is stored. It cannot be modified.

security out of service switch (directories)

When this switch is on, the directory in which it occurs and all inferior segments and directories 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 (segments, directories, links)

The type attribute indicates whether an entry refers to a segment, a directory, or a link. The type attribute cannot be modified.

unique identifier (segments, directories, links)

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.

Multisegment Files

Very large data bases may need to 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 termed 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, e.g., names; however, due to the nature of an MSF, 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 switch 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 MPM Subroutines.

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 programmer should consult the individual command or subroutine description before invoking it on an MSF.

SYSTEM DIRECTORIES

A single directory hierarchy is used for both system and user segments. Figure 2-2 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.

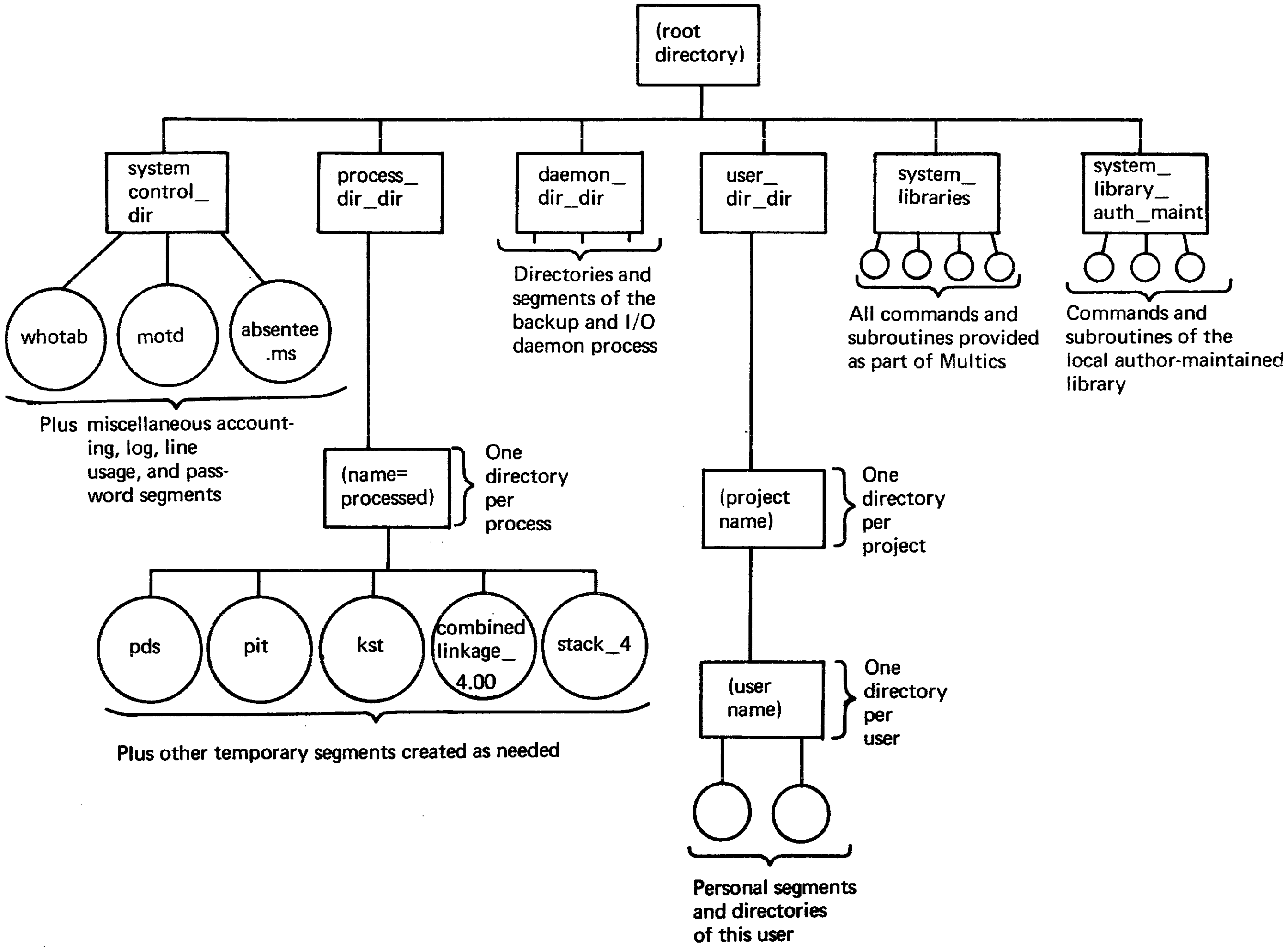

(root directory)
system control_ dir
process_ dir_dir
daemon_ dir_dir
user_ dir_dir
system_ libraries
system_ library_ auth_maint
whotab
motd
absentee .ms
Plus miscellaneous accounting, log, line usage, and password segments
Directories and segments of the backup and I/O daemon process
All commands and subroutines provided as part of Multics
Commands and subroutines of the local author-maintained library
(name= processed)
One directory per process
(project name)
One directory per project
pds
pit
kst
combined linkage_ 4.00
stack_4
Plus other temporary segments created as needed
(user name)
One directory per user
Personal segments and directories of this user

Figure 2-2. Directory Hierarchy

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

1. system_control_1

This directory is used for administrative purposes. It 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 entries for a set of directories, each of which is associated with a 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 five initial segments described below:

process data segment (PDS)
: A supervisor data base, the PDS keeps a record of the state of the process. This segment is 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.

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, except in the case of ring 0 where the PDS is used as a stack.

combined_linkage_n.mm
: This segment, managed by the 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 the same name, the last two characters (mm) serving as a sequence number, beginning with 00 for the first segment. In addition, each active ring has its own linkage segment, where the ring number is indicated by the character (n) preceding the period. Thus, combined_linkage_3.02 indicates the third linkage segment for ring 3.

Other segments commonly found in the process directory include the free storage area used to implement PL/I allocate and free statements, the parse tree of the PL/I compiler, and temporary storage areas used by Multics editors. These segments are created as needed.

3. daemon_dir_dir

This directory catalogs segments that support system daemon processes, such as automatic file backup and bulk (card and printer) input and output. Only one portion of the entries in this directory is generally accessible to users -- the queues of the I/O facilities.

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 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 three system libraries:

system_library_standard
system_library_1
system_library_tools

The procedures in these directories are documented in the MPM Commands, MPM Subsystem Writers' Guide, and MPM Subroutines. A library of unbundled software (system_library_unbundled) may also be present. Unless the user specifies otherwise, these directories are included in the list of directories to be searched during dynamic linking. See "Dynamic Linking" and "Search Rules" in Section IV for descriptions of dynamic linking and search rules.

6. system_library_auth_maint

This directory is similar to the standard system libraries except that it contains commands and subroutines provided by programmers of the local installation.

SECTION III

NAMING, COMMAND LANGUAGE, AND TYPING CONVENTIONS

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, and I/O switch names. User names are discussed under "Access Control" in Section VI since they are primarily used to specify access control information.

Entrynames

An entryname is the name of a segment, multisegment file, directory, or link. An entryname consists of at least one nonblank and no more than 32 ASCII characters. Any entry (segment, multisegment file, directory, or link) 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. 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, such as, square_root.pl1 for the name of a PL/I source segment. See "Program Preparation" in Section IV for details on programming conventions.

Only the greater-than (>) character is prohibited in entrynames, since it is used to form pathnames as described below. Several other characters are not recommended for entrynames -- less-than (<), asterisk (*), question mark (?), percent sign (%), equal sign (=), dollar sign ($), quotation mark ("), and parentheses -- because standard commands attach special meanings to them. 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 II.) The last entryname in a pathname is the name of a segment, multisegment file, directory, or link entry. 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, or a link. 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.

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

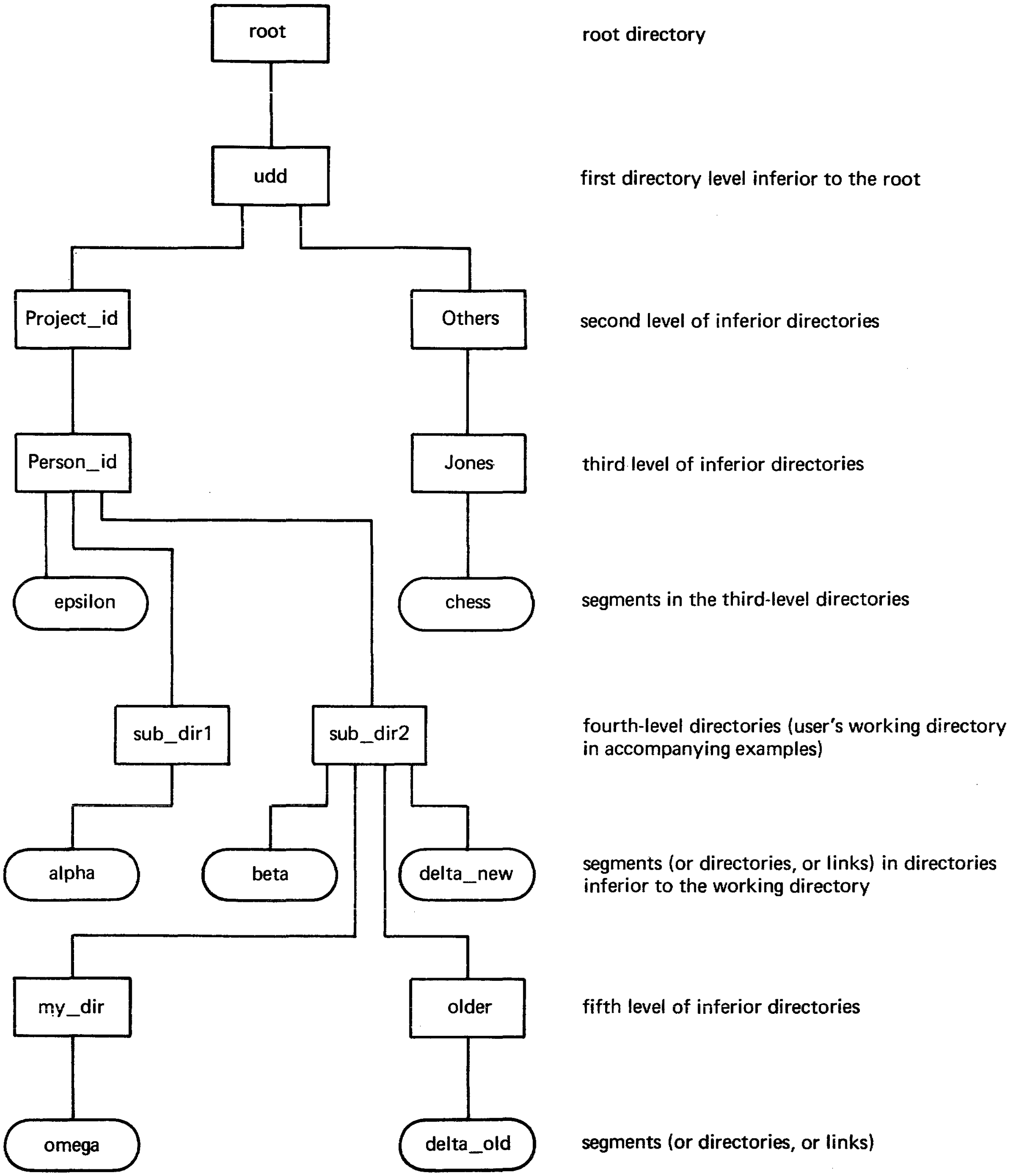

root
root directory
udd
first directory level inferior to the root
Project_id
Others
second level of inferior directories
Person_id
Jones
third level of inferior directories
epsilon
chess
segments in the third-level directories
sub_dir1
sub_dir2
fourth-level directories (user's working directory in accompanying examples)
alpha
beta
delta_new
segments (or directories, or links) in directories inferior to the working directory
my_dir
older
fifth level of inferior directories
omega
delta_old
segments (or directories, or links)

Figure 3-1. Sample Storage Hierarchy

Star Names

Many commands that accept pathnames as input allow the final entryname in the pathname to be a star name. A star name is an entryname. The star convention matches a star name with entrynames in a single directory to identify a group of entries. The entrynames matching a star name have components in common and are matched according to specific rules. Commands that accept star names perform their function on each entry identified by the star name.

CONSTRUCTING STAR NAMES

Star names are constructed according to the following rules:

1. A star 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, a star name cannot contain control characters such as backspace, tab, or newline.

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

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

4. Each asterisk or star (*) character (referred to as a star) appearing in a star name component is treated as a special character.

5. A star name component consisting only of a double star (**) is treated as a special component.

INTERPRETING STAR NAMES

A star name is matched to entrynames in a single directory according to the following rules:

1. Each question mark (?) in a star name component matches any one character that appears in the corresponding component and character position of an entryname.

2. Each asterisk (*) in a star name component matches any number of characters (including none) that appear in the corresponding component and character position of an entryname. If the asterisk is the only character of the star name component, it matches any corresponding component of an entryname. Only one asterisk can appear in each star name component, except for the double star component as noted in the next rule.

3. The double star component (**) in a star name matches any number of components (including none) in the corresponding component position of an entryname. Only one double star component can appear in a star name.

The rules above do not require that star names contain asterisks or question marks. Therefore, an entryname that does not contain either of these special characters can be used as a star name, as long as it does not contain any null components (two consecutive periods make a null component). When such an entryname is used as a star name, the directory is searched until the single entryname that matches is found. The rules above impose no restrictions on the form of the entrynames to be matched with the star name. Such names can contain null components that match only star name components of * or **.

The following examples illustrate some common forms for star names.

!??????????????
identifies all fifteen character one-component entries beginning with ! (called unique names because such names are generated by the unique_chars_ subroutine, described in the MPM Subroutines, 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, such that the first component has exactly five characters and begins with prog, and the last component is pl1.

Equal Names

Some commands that accept pairs of pathnames as their arguments (e.g., the rename command described in the MPM Commands) allow the final entryname of the first pathname to be a star name, and the final entryname of the second pathname to be an equal name. An equal name is an entryname containing special characters that represent one or more characters from the entrynames identified by the star name (not characters from the star name itself). 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 the entrynames identified by the 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 equal sign (==) is treated as a special component.

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 an entryname identified by the star name. An error occurs if the corresponding character does not exist.

2. An equal sign (=) in an equal name component represents the corresponding component of an entryname identified by the star name. 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 equal sign component, as noted in the next rule.

3. The double equal sign (==) component of an equal name represents all components of an entryname identified by the star name that have no other corresponding components in the equal name. Often, the double equal sign component represents more than one component of an entryname identified by the star name. 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.

The rules above do not require that equal names contain equal signs or percent characters. Therefore, an entryname that does not contain either of these special characters can be used as an equal name, as long as it does not contain any null components. Also, 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 an entryname that contains null components, because these commands treat their arguments as either star names or equal names. The fs_chname command can be used to rename entries if names containing null components are accidentally created. (See the MPM Commands for a description of the rename, add_name, and fs_chname commands.)

The following examples illustrate how equal names might be used in rename and add_name commands. 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  alpha  beta.=.gamma
```

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

The command:

```
     rename  program.pl1  old_=.=
```

is equivalent to:

```
     rename  program.pl1  old_program.pl1
```

and the command:

```
     add_name  data  first_=_set
```

is equivalent to:

```
     add_name  data  first_data_set
```

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, and the first components (if any) are the same as those of the name ending in ec (e.g., the name alpha.absin would be added to the entry named alpha.ec). 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.

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 IV and "Process" in Section I.) A segment can be made known via the initiate command (described in the MPM Commands) and the hcs_$initiate and hcs_$initiate_count subroutines (documented in the MPM Subroutines). 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 MPM Subroutines) uses search rules to locate the desired segment. The reference name table, listing reference names for segments, is always 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 IV.)

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 MPM Commands and the term_, hcs_$terminate_file, and hcs_$terminate_seg subroutines in the MPM Subroutines.) 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 will 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 MPM Commands and the term_, hcs_$terminate_name, and hcs_$terminate_noname subroutines in the MPM Subroutines.) 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>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 MPM Commands) 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 MPM Subroutines) 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);
```

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 one of his procedure segments (perhaps to preserve an old copy) or created a storage system link to a segment using a different name, he must thereafter use the full reference name/entry point name construction 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.

1. Each is permitted to be 32 characters or less in length.
2. All ASCII characters are legal in any position except as noted in the following points and "Entrynames" above.
3. System subroutine names end in an underscore to prevent conflicts with subroutine names given by users; that is, the user can easily avoid conflicts by not having an underscore as the last character of any of his subroutine names.

4. 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 Appendix E, "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.

5. Command and subroutine names should not contain a period; i.e., they should have only one component.

COMMAND LANGUAGE

A command procedure is a special type of PL/I procedure that uses argument processing facilities provided by the command processor. A Multics system command invocation consists of a command procedure name (command name) plus any character-string arguments to be passed to the command procedure when the command is invoked. A Multics system command invocation consists of a command procedure name (command name) plus any character-string arguments to be passed to the command procedure when the command is invoked. Most user programs that take character strings as arguments can be invoked as commands. (See "Writing a Command" in Section IV.) The Multics command processor is a mechanism for invoking programs by command name. It is called by the Multics listener subroutine to process the command lines 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 MPM Subroutines.)

A command line contains one or more commands separated by semicolons (;). The syntax of a command line should not be identical to the subroutine call of a programming language. Calls in most programming languages are cumbersome to type. Instead, the conventions for the syntax of a command line are chosen for simplicity in the basic case, and for functional flexibility otherwise. The command language provides various services such as nesting and iteration of commands. The command language syntax allows these features to be specified by means of certain special delimiters in the command line; but if the services are not desired, the user need only type his commands according to the format discussed below under "Simple Commands." The special services of the command system are then bypassed.

Some subsystems under Multics may choose to interact with their users with their own conventions. The following description does not apply to these subsystem command languages.

Command Environment

After a user successfully logs in to the system, the listener prints a ready message on the terminal. The user is now at command level and the system is available for new commands. When a command has finished executing, it returns to the listener, which prints a ready message. At command level, the user issues commands in the syntax of the command language described below. Multics terminal input allows read-ahead; therefore, the user does not have to wait for a ready message before typing another command line. The user can, however, be interrupted in the middle of typing a line by the ready message or by output printed by the command. If this occurs, some characters in the line being typed may be lost. 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 MPM Commands).

Simple Commands

A command specifies a function to perform and, if necessary, the arguments with which the function operates. Command names and command arguments are treated as character strings. Individual commands convert numeric characters to binary representation, as needed.

The command consists of two basic elements: the command name and the arguments. The command name is essentially a reference name, and if appropriate, an entry point name. The command processor uses the user's search rules (see "Search Rules" in Section IV) to find the program whose name is the command name. A pathname may be used in place of the reference name to override the 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, as described above. Notice, that since the segment has been initiated with a reference name, it can be identified by reference name in subsequent commands (see example below). The argument portion of the command is simply character strings designating, for instance, a segment. The number of arguments, if any, depends on the command invoked. The elements of a command (command name and command arguments) are delimited by the space (also called a blank). The terminator of a command can be either the semicolon (;) or the newline character. More than one command can be issued on the same line of terminal input by using the semicolon between commands.

The general form of the simple command is:

```
command_name control_args
```

where each element is separated from the preceding one by one or more spaces. For example, the rename command takes arguments in pairs; the first 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 procedure named rename at entry point rename with the character strings square_root and sqrt as arguments. There is no difference between invoking a command from the terminal and calling it in a procedure. For example, typing the command line "rename square_root sqrt" is equivalent to executing the following PL/I program:

```
x: proc;
call rename ("square_root", "sqrt");
end x;
```

As another example, suppose a user knows that an experimental version of the rename command resides in the directory >Smith_dir. If the user types:

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

then the experimental version is invoked instead of the version that would have been found by the search rules. Subsequent unqualified references to the rename command invoke the one in >Smith_dir. Any program in the storage system hierarchy can be invoked in the same way.

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 (|) when adjacent to the left bracket. Occasionally, however, it is necessary to use a reserved character without its special meaning. For example, a user might want to pass a semicolon as an argument to a command. The quotation mark character (") is reserved for this purpose; i.e., reserved characters within a quoted string (i.e., a string of characters surrounded by quotation marks) are treated as ordinary characters. Thus:

```
rename ";" foo
```

causes a semicolon to be passed as an argument to the rename command. Also, since a quotation mark is a reserved character, it may be desirable to suppress its special meaning. For this purpose, two adjacent quotation marks within a quoted string are interpreted as a single quotation mark. For example:

```
delete "A""B"
```

causes the argument A"B to be passed to the delete command.

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 one or more elements enclosed by parentheses. Each element of the set, in turn, replaces the entire iteration set in the command line. For example:

```
print (a b c).pl1
```

is equivalent to the three commands:

```
print a.pl1; print b.pl1; print c.pl1
```

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)
```

would expand 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 the outside in. 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)
```

would create three directories:

```
     >Smith_dir>new>first

     >Smith_dir>new>second

     >Smith_dir>old>third
```

The ability of the Multics command language to perform concatenation underlies the iteration feature. See "Concatenation" below.

Active Strings

An active string is defined to be a part of a command that is immediately evaluated (executed) and the resulting value placed back into the command line. A program explicitly designed to be used in an active string is called an active function. (See the MPM Commands for a description of the active functions available on Multics.) An active function must return a varying character string as its value.

The delimiters of an active string are the left bracket ([) and the right bracket (]). The following example illustrates the use of active strings in a command.

```
     delete  [oldest_segment]
```

This command returns its value as a varying character string rather than printing its value on the terminal. The command processor does the following: scans the command line; discovers the active string; and evaluates it, placing the obtained value into the command line. The command processor then: discovers the terminator (a newline character); and evaluates and executes the remainder of the command, finding the command name (delete) and the character string returned by the oldest_segment active function as an argument.

Active functions can have arguments of their own, and active strings can be nested. For example, if there were a random name generating routine called namer that takes an arbitrary two-character string as a seed and returns a varying character string 32 characters or less in length, the command line:

```
rename [oldest_segment] [namer xz]
```

causes the least recently used segment to be renamed to whatever random name emerged from namer when the latter was invoked with xz as an argument. If there were a random number generator called random for priming namer, the command line:

```
rename [oldest_segment] [namer [random]]
```

is also valid, provided random returns a varying character-string value.

After an active string is evaluated, the value returned is rescanned for active strings before being inserted into the command line. For example, if procedure alpha:

```
x [alpha]
```

returned [beta] as its value and x were the name of a command, procedure beta would be invoked as an active function. The x command would be invoked with whatever value procedure beta, in turn, returned.

The user may suppress rescanning of the returned string for further active strings by placing a vertical bar (¦) before the active string. For example:

```
x ¦[alpha]
```

results in the invocation of x with [beta] as an argument. All other scanning (e.g., for spaces) is performed on the returned string.

Iteration can, of course, be combined with the above features. For example, if the program "bill" returned the character string "arthur robert fred", the command line:

```
print ([bill])
```

expands into:

```
print (arthur robert fred)
```

which would be expanded into:

```
print arthur; print robert; print fred
```

The command line finally obtained when all active strings have been processed is called the expanded command line. The maximum length of the expanded command line is, by default, 128 characters. This size can be changed using the set_com_line command (described in the MPM Commands). For efficiency, it is recommended that the size be left at 128 characters except when a larger size is temporarily needed (e.g., to accommodate a large returned string from some active function).

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. The value of the active string is the concatenation of the values of the active functions. For example, if the active string:

```
     [act_fnc1 x]
```

returns the value TURN, and the active string:

```
     [act_fnc2 y z]
```

returns the value OUT, then the active string:

```
     [act_fnc1 x; act_fnc2 y z]
```

returns the value TURNOUT.

Concatenation

The Multics command language has the ability to form basic elements (e.g., character strings) by concatenation with nonbasic elements (e.g., the values of active strings). For example, active function home_dir returns the character string representation of the pathname of the user's home directory. Therefore, it is possible to perform 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. In the Multics command language, this facility is furnished in precisely the manner shown. That is, the value of a delimited element of a command is concatenated with the string or delimited element adjacent to it when there is no space between the two.

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

```
     delete  [home_dir]>([bill])
```

deletes the segments arthur, robert, and fred in the user's home directory where the active function, bill, is defined as above.

Concatenation is permissible in either direction with regard to the delimited string and the nondelimited string. For example:

```
     delete >project_dir>Doe>([bill])
```

deletes the segments arthur, robert, and fred in the directory >project_dir>Doe.

TYPING CONVENTIONS

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

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 key strokes, 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. 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 subroutine is accessed via a call to the iox_ subroutine. (See the descriptions of the tty_ I/O module and the iox_ subroutine in the MPM Subroutines.) 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   the specified data is read 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 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 "≷̲≷̲" is always stored as:

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

Examples of Canonical Form

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 f̲u̲l̲l̲ 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.

Erase and Kill Characters

Two minimal editing capabilities on the line currently being typed are available. They are:

1. the ability to delete the latest character or characters.
2. the ability to delete all of the current line.

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

The first editing convention reserves one graphic, the number sign (#), as the erase character. 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 graphics. One erase character typed immediately after "white space" causes the entire white space to be erased. (Any combination of tabs and spaces is called a 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
```

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, the commercial at sign (@), as the kill character. 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 a number sign.

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

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@The offBBB___##nB_ stateN
Typed line:      This@The off##n state
Final input:     The _Bo_Bn stateN
Printed line:    The on state
```

ASCII CHARACTER SET

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. See Appendix A for a table of the ASCII character set, a list of printing graphic characters, control characters, and unused characters.

Escape Characters

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 character. 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 standard escape 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 d*i* 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.

6. The solid vertical bar (|) and the broken vertical bar (¦) are equivalent representations of the graphic corresponding to ASCII code 174.

SECTION IV

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 presented in this section presents two basic aspects of the programming environment. One of these, program preparation, presents the steps involved in implementing a program to run on Multics. The remainder of the section presents the major internal interfaces between a user program and the system that are automatically or explicitly activated during program execution.

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	proposed American National Standards Institute (ANSI) standard PL/I
FORTRAN	superset of ANSI standard FORTRAN
COBOL	subset of the ANSI standard COBOL
BASIC	compatible with the Dartmouth Version 6 BASIC
ALM	Multics assembly language
APL	interactive interpreter (based on IBM APL)

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. Programs written in different languages produce compatible object segments. 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 edm or qedx. (See the MPM Commands 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
```

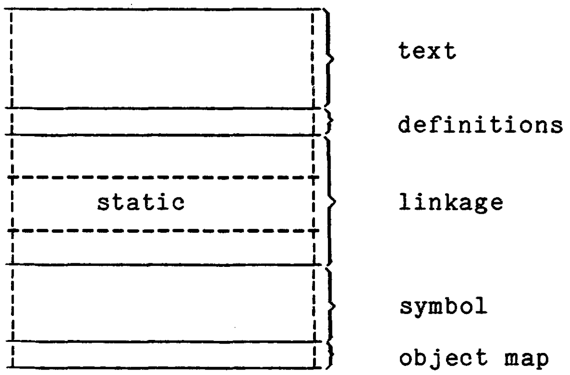
and an object segment named square_root are produced and placed in the user's working directory.

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 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 without this 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. linkage — is a prototype linkage section, containing PL/I internal static variables, if any, as well as dynamic linkage information. The linkage section is copied into a per-process data base (the combined linkage region) when the object segment is first referenced and contains information used by the dynamic linker to resolve external references.

4. 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.

5. 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 "Multics Standard Object Segment" in the MPM Subsystem Writers' Guide.

Debugging Facilities

Multics provides extensive interactive program debugging facilities through the two commands, debug and probe. To perform symbolic debugging with these commands, the user must have compiled his program with a symbol table (i.e., specifying the -table control argument). 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 MPM Commands.)

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. He may then continue execution from the point at which execution 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 adjustable, 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 an arbitrary 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 the fifth input argument to the procedure and reports an error to the user if the argument was not given in the command line.

```
sample_command: procedure;
declare argl fixed binary(17);
declare argp pointer;
declare arg  character (argl) based (argp);
declare code fixed binary(35);
declare cu_$arg_ptr entry (fixed binary(17), pointer,
     fixed binary(17), fixed binary(35));
declare com_err_ entry options(variable);
     .
     .
     .
call cu_$arg_ptr (5, argp, argl, code);
if code ^= 0 then do;  /* expected arg missing  */
     call com_err_ (code, "sample_command",
          "Five arguments are required.");
     return;
     end;
if arg = "-input" then
     .
     .
     .
end sample_command;
```

Detailed information about the command utility and command error subroutines is provided in the MPM Subroutines.

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 adjustable, 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 an arbitrary 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(17);
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(17), 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 MPM Subroutines.

The same procedure may 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 if they were an active function. However, if an error code is returned by cu_$af_return_arg, they operate as a command.

ADDRESS SPACE MANAGEMENT

When a user logs in, he 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, 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 III 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 "Multics Standard Object Segment" in Section I in the MPM Subsystem Writers' Guide.) 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 MPM Commands).

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_$initiate, hcs_$initiate_count, or hcs_$make_seg with a nonnull character string supplied as the ref_name argument (these hcs_ entry points are described in the MPM Subroutines)

   c. a call to hcs_$make_ptr (described in the MPM Subroutines)

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 MPM Commands and MPM Subroutines 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.

>system_library_auth_maint

This library contains the author-maintained and installation-maintained libraries. The author-maintained library consists of a collection of procedures contributed by users at a particular installation. It is maintained for the convenience of the local user community, as an aid in sharing programs. Users of author-maintained procedures should be aware of two things:

a. there may have been little or no verification of the effectiveness or accuracy of the procedures

b. no guarantee is made that the procedures will continue to be maintained as the system changes (especially a problem in the case of language translators)

The installation-maintained library contains procedures installed and maintained by the local installation. It differs from the author-maintained library in that verification of accuracy and effectiveness of the procedures has been performed by the installation, and the installation is committed to maintaining the procedures.

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 MPM Commands; for the structure of a bound segment, see the "Structure of Bound Segments" in Section I in the MPM Subsystem Writers' Guide.)

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 to the hcs_$initiate or hcs_$initiate_count subroutines (described in the MPM Subroutines).

When a segment is made known, a reference name can also be associated with it. Such a name is said to be initiated for the segment. 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. Reference names can also be initiated using the hcs_$initiate or hcs_$initiate_count subroutines.

The user can remove reference names by using the hcs_$terminate_name subroutine (described in the MPM Subroutines). If only one reference name appears for a segment and it is terminated, the segment is also made unknown. The user may also explicitly make a segment unknown and terminate all its reference names (see hcs_$terminate_file and hcs_$terminate_seg in the MPM Subroutines).

At command level, the initiate and terminate commands may be used to initiate and terminate reference names. (See the MPM Commands 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 hcs_$initiate.

A brief description of these interface subroutines is given below. For a complete description, see the MPM Subroutines.

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

hcs_$initiate
hcs_$initiate_count — given a pathname and, optionally, a reference name, causes the segment to be made known and the reference name, if supplied, to be initiated. If hcs_$initiate_count is used, a bit count of the segment is also returned.

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.

hcs_$terminate_file — given a pathname, terminates all reference names of a segment and makes it unknown.

hcs_$terminate_name — terminates one reference name from a segment. If it is the only reference name for that segment, the segment is made unknown.

hcs_$terminate_noname — given a pointer to a segment, makes the segment unknown if there are no reference names associated with the segment.

hcs_$terminate_seg — 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 Section II of the MPM Subsystem Writers' Guide.

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:

1. Resource monitoring and accounting.
2. Labeling data (e.g., storage system entries) with dates and times of interest.
3. Computing the date and time for output.
4. Generating a unique bit string.

5. Waking up a specified process at a specified time, perhaps causing a specified procedure to be called.

6. 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 MPM Subroutines. The clock_ subroutine reads the real-time clock and returns 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 value of the process execution timer is returned by both the cpu_time_and_paging_ and virtual_cpu_time_ subroutines. The resource_usage command (described in the MPM Commands) 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 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 MPM Subsystem Writers' Guide) 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 MPM Subsystem Writers' Guide) 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.

SECTION V

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, and Bulk I/O. Appendix C ("Punched Card Input and Output") explains how to use the punched card facilities available on Multics. Related documentation is referenced where necessary.

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 MPM Subroutines). 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 MPM Commands.

System Input/Output Modules

The Multics system contains the following I/O modules:

discard_ is a sink for unwanted output

rdisk_ supports I/O from/to removable disk packs

record_stream_ provides a mechanism for doing record I/O on an unstructured file, or vice versa.

syn_ establishes one switch as a synonym for another

tape_ansi_ supports I/O from/to magnetic tape files according to standards proposed by the American National Standards Institute (ANSI)

tape_ibm_ supports I/O from/to magnetic tape files according to standards established by IBM

tape_mult_ supports I/O from/to magnetic tape files in Multics standard tape format

tty_ supports I/O from/to terminals

vfile_ supports I/O from/to files in the storage system

These modules are described in Section III of the MPM Subroutines.

User-Written Input/Output Modules

The user may construct his own I/O system interface modules. See "Writing an I/O Module" in Section IV of the MPM Subsystem Writers' Guide.

How to Perform Input/Output

To perform the 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 MPM Subroutines) or by use of the io_call command (described in MPM Commands). The I/O facilities of the programming languages may also be used to carry out these steps.

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 MPM Subroutines). This attachment could also have been performed by a subroutine call as follows:

   ```
   call iox_$attach_ioname ("input_sw", iocb_ptr,
        "vfile_ some_file", code);
   ```

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. Example:

   ```
   call iox_$open (iocb_ptr, 4, "0"b, code);
   ```

   The iocb_ptr identifies the switch (see. "Input/Output Switches" below). The argument 4 means that the opening is for sequential reading. The "0"b represents an obsolete argument. See the description of the iox_ subroutine (described in the MPM Subroutines) for full details. This open step could also have been performed by a command, as follows:

   ```
   io_call open input_sw sequential_input
   ```

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);
   ```

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. Example:

   ```
   call iox_$close (iocb_ptr, code);
   ```

5. Detach the I/O switch. After this step, the switch can be attached again for some other purpose. Example:

   ```
   io_call detach input_sw
   ```

In general, only step 1 (attach) involves peculiarities of a particular type of device or a particular file format. It is often convenient to have this step and step 5 (detach) performed from command level, while steps 2 to 4 are performed by a program. This may be used to make a program device independent.

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 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_$attach_iocb or iox_$attach_ioname entry points should be invoked. (See the MPM Commands and the MPM Subroutines, respectively.) In all cases, an attach description must be given. This string has the following form:

```
module_name -options-
```

where module_name and each option do not contain blanks but are separated from one another by one or more blanks.

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 Section III of the MPM Subroutines. 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 MPM Commands) may not be exactly the same as the attach description given to the io_call command or the iox_$attach_iocb or iox_$attach_ioname 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 or the iox_$open entry point are used to open a switch. 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.

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 commands 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. The result of these four commands is that output that would normally be sent to a terminal is written into a file. The file_output command (described in the MPM Commands) performs this sequence of steps and is the normal way of directing terminal output to a file.

```
io_call attach file_switch vfile_ file_name -extend
io_call open file_switch stream_output
io_call detach user_output
io_call attach user_output syn_ file_switch
```

The following commands can be used to undo the effects of the previous four commands with the result that subsequent output to the I/O switch user_output is written on the user's terminal. The console_output command (described in the MPM Commands) performs this sequence of steps and is the normal way of reverting user_output back to its normal attachment (the terminal).

```
io_call detach user_output
io_call attach user_output syn_ user_i/o
io_call close file_switch
io_call detach file_switch
```

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	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	x	x	x			x		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.

Table 5-2. Opening Modes Supported by I/O Modules

Opening Mode		I/O Module							
No.	Name	discard_	rdisk_	record_stream_	tape_ansi_	tape_ibm_	tape_mult_	tty_	vfile_
1	stream_input			X			X	X	X
2	stream_output	X		X			X	X	X
3	stream_input_output							X	X
4	sequential_input			X	X	X			X
5	sequential_output	X		X	X	X			X
6	sequential_input_output								X
7	sequential_update								X
8	keyed_sequential_input								X
9	keyed_sequential_output	X							X
10	keyed_sequential_update								X
11	direct_input		X						X
12	direct_output	X							X
13	direct_update		X						X

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.

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.

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 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 the user's terminal is:

```
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 module descriptions in Section III of the MPM Subroutines 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 control blocks 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 MPM Commands) closes PL/I and FORTRAN control 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.

FILE INPUT/OUTPUT

The I/O system distinguishes three types of files: unstructured, sequential, 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 lines

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

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

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. 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 comparision (lexicographic order using the ASCII collating sequence).

All the I/O operations applicable to sequential files apply to indexed files as well. 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)

Table 5-3 shows the I/O operations that are permitted with each type of file.

File Opening

When an I/O switch is attached to a file and is opened for input or update, 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 or input_output, and the file does not exist, a file of the appropriate type is created. 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 either the attachment or the opening 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.

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.

The default handler for the finish condition closes all I/O switches.

Table 5-3. File Types and Allowed Input/Output Operations

	Input/Output Operation										
Type of File	get_line	get_chars	put_chars	read_record	rewrite_record	delete_record	read_length	position	seek_key	read_key	write_record
unstructured (sequence of 9-bit bytes, usually ASCII characters)	X	X	X					X			
sequential (sequence of records)				X	X	X	X	X			X
indexed (sequence of records and an index)				X	X	X	X	X	X	X	X

Each record is a string of bytes; a record may be of zero length. For an indexed file, a key is a string of 0 to 256 ASCII characters, with no trailing blanks.

Table 5-4. Compatible File Attachments

Opening Mode		File Type		
No.	Name	unstructured	sequential	indexed
1	stream_input	x	1	1
2	stream_output	x3		
3	stream_input_output	x3		
4	sequential_input		x	x
5	sequential_output		x3	
6	sequential_input_output		x3	
7	sequential_update		2	x
8	keyed_sequential_input			x
9	keyed_sequential_output			x3
10	keyed_sequential_update			x
11	direct_input			x
12	direct_output			x3
13	direct_update			x

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 opening for the specified mode without using the -extend control argument. (See the individual I/O module descriptions in Section III of the MPM Subroutines to see if the -extend control argument is applicable.)

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 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.

TERMINAL INPUT/OUTPUT

Terminal I/O is most conveniently done through the appropriate programming language facilities. Since each language handles I/O in a different manner, the user should consult the reference manual and/or user's guide of the language in question.

The set_tty command may be used to control characteristics of terminal I/O such as the line length, insertion of tabs, and erase and kill processing. See the descriptions of the set_tty command in the MPM Commands and the tty_ I/O module in the MPM Subroutines.

The file_output command causes all subsequent output normally printed on the user's terminal to be written instead to a file in the storage system. The console_output command causes such output to be directed again to the terminal. Both of these commands are described in the MPM Commands.

The contents of segments that contain ASCII characters may be printed on the terminal by invoking the print command. The contents of any segment can be printed in octal using the dump_segment command or the Multics debug command. See the descriptions of print, dump_segment, and debug in the MPM Commands.

The ioa_ subroutine provides a convenient means for formatting output to be printed on the terminal, and it may be used for other output as well. See the ioa_ subroutine description in the MPM Subroutines.

Table 5-5. File Position Designators at Open

Opening Mode		Designator			
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			
4	sequential_input		first record		
5	sequential_output				
6	sequential_input_output		end of file		
7	sequential_update		first record	null	
8	keyed_sequential_input		first record		
9	keyed_sequential_output				null
10	keyed_sequential_update		first record	null	null
11	direct_input				
12	direct_output				null
13	direct_update			null	null

In the opening where no value is indicated for a designator, the designator is not relevant.

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 dprint 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 dprint command in the MPM Commands.

The printed output has the following parts:

1. Header sheet. This sheet identifies: the requesting User_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 4) 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 is 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 dprint command. See the dprint command in the MPM Commands 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; the number of blocks and the cost per 1000 blocks (a block is the bit count of the file divided by 750); the total cost of the output and the User_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.

Punched Card Input

Facilities are provided to read punched card decks into Multics files. There are three types of card formats that can be input to Multics: Multics card codes, 7punch, and raw.

mcc — The Multics card codes are defined in "Punched Card Codes" in Appendix C of this document. They consist 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. Trailing blanks on a card are ignored. A newline character is inserted after the end of each card.

7punch — The 7punch decks are binary representations of existing files, 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. 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.

Punched Card Output

The dpunch command described in the MPM Commands causes the contents of Multics files to be punched. The files can be punched under mcc, raw, or 7punch conversion modes. See Appendix C for more information on punched card output.

SECTION VI

ACCESS CONTROL

"Access control" means regulating how a process is allowed to use or refer to information within the system. Every process is executing on behalf of some user (and has associated with it the name of this user). Access rights are regulated in terms of a particular process or a particular user. That is, access rights are thought of as being granted on a per-user or a per-process basis depending on the type of control being discussed.

TYPES OF ACCESS CONTROL

There are three types of access control on Multics: discretionary access control, which is regulated by an access control list (ACL); nondiscretionary access control, which is regulated by the access isolation mechanism (AIM); and intraprocess access control, which is regulated by the ring structure. Each type of access control is briefly described as follows:

Discretionary access control
: allows individual users to grant or deny other users access to their segments and directories at their own discretion. ACLs, which regulate this type of access, are of interest primarily to users who wish to share their programs and data bases with other users.

Nondiscretionary access control
: enforces the policies of the system administration and of the organizations served by the system. (This type of access control is sometimes referred to as administrative access control.) By defining access authorizations for processes and access classes for directories and segments, the system administration (through AIM) guarantees that only authorized persons may access certain classes of information. In general, nondiscretionary controls are used to restrict discretionary controls; for example, with AIM, users cannot "give away" access to information or programs even though they themselves have complete access to the data.

Intraprocess access control
: provides the ability for programs to enforce arbitrary access control policies that go beyond the basic ACL and AIM controls. For example, the ring structure protects the supervisor programs from actions of users. Rings are useful for those persons writing subsystems (containing programs and data bases that will be used by many users) that require more specifically defined protection than is offered by the other access controls.

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 an object by discretionary access control (the ACL) is known as the process' raw access mode or raw mode. Since the several access control mechanisms operate together, the access modes of a process to an item at any instant of time are those granted by all controls. For example, if the discretionary controls grant a process read and write modes to a segment, but the nondiscretionary (or administrative) controls allow the process read mode, the process may only read the segment, but may not store into it. Thus, any of the other controls can restrict, but not extend, the access granted by the discretionary controls. The actual access mode that the system enforces for each reference or use of a segment or directory is called the process' effective access mode to that segment or directory.

DISCRETIONARY ACCESS CONTROL

Discretionary access controls allow individual users to grant access to directories and segments on a per-user basis. Users can exercise discretionary control only within those portions of the storage system hierarchy where they themselves have the proper access rights.

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 process has an associated name called an access identifier. The access identifier is fixed for the life of the process. The identifier is a three-component character string that must be less than 33 characters where the first component is the registered name of the person on whose behalf the process was created (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 of "m". The instance tag of "z" is used for certain system processes, e.g., one that runs a line printer. The access identifier Jones.Mentor.a would be associated with an interactive process created for Jones on the Mentor project.

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

The access rights a process has to segments and directories are described by access modes. These modes allow a user to stipulate several different kinds of access on the same item in the storage system if he so desires.

The access modes for segments and directories are listed below. The single letter in the first column is used by the various access-related system commands to indicate the respective mode.

The access modes for segments are:

r read the process can execute instructions that cause data to be fetched (loaded) from the segment.

e execute an executing procedure can transfer to this segment and words of this segment can then be interpreted as instructions and executed by a processor.

w write the process can execute instructions that cause data in the segment to be modified.

n null the process cannot access the segment in any way.

The access modes for directories are:

s status the attributes of segments, directories, and links contained in the directory and certain attributes of the directory itself can be obtained by the process (for a definition of attributes, see "Segment, Directory, and Link Attributes" in Section II).

m modify the attributes of existing segments, directories, and links contained in the directory and certain attributes of the directory itself can be modified; and existing segments, directories, and links contained in the directory can be deleted.

a append new segments, directories, and links can be created in the directory.

n null the process cannot access the directory in any way.

Null access is implied by default; that is, if a user does not issue a command to grant other users access to some segment or directory, these other users cannot access the data in any way.

The access modes described above pertain to all entries in the storage system. In addition, another group of access modes, known as extended access, applies to both types of message segments, queue and mailbox. Queue message segments contain I/O requests (from the dprint and dpunch commands) and absentee requests. Mailbox message segments contain messages sent by many different users. Extended access provides a way to further control operations on message segments and ensure privacy among users.

The extended access modes for mailboxes are:

a	add	the process can add a message
d	delete	the process can delete any message
r	read	the process can read any message
o	own	the process can read or delete only its own messages; that is, those sent by this process
s	status	the process can find out how many messages are in the mailbox
n	null	the process cannot access the mailbox in any way

Access on a newly created mailbox is automatically set to adros for the user who created it, ao for *.SysDaemon.*, and ao for *.*.*. Access on queue message segments is controlled by the site, but is usually set to aros for *.*.*.

Structure of an ACL

The rights that different user processes have when referring to a segment or directory are specified as an attribute of that segment or directory in the form of a list called the access control list (ACL). Each entry of the list specifies a set of processes (actually a set of access identifiers of processes) and the access modes that members of that set can use when referring to the segment or directory. The modes read, write, execute, and null can be specified in ACLs of segments and the modes status, modify, append, and null can be specified in ACLs of directories. On directory ACLs, modify mode cannot appear without status mode. On segment ACLs, write and execute modes cannot appear without read mode.

Only those access modes actually granted in an ACL entry are given to the specified user process. For example, if the ACL of a segment contains the modes read and execute for a specific user, then that user's process can fetch data from the segment and transfer to and execute instructions in the segment, but it cannot modify data in the segment (because the write access mode was not granted).

Each ACL entry designates certain access modes and identifies the users that have these access modes. Since each user process is identified by an access identifier when the process is created (explained above), it would seem logical to use that identifier to designate a user in the ACL entry. However, if access identifiers were used to specify every user process that could access a segment or directory, the ACLs would become extremely cumbersome.

For example, to grant read and execute access on a particular segment to all six members of the Demo project, whether they are using an interactive or absentee process, would require 12 separate ACL entries:

```
Jones.Demo.a   re
Jones.Demo.m   re
Smith.Demo.a   re
Smith.Demo.m   re
      .
      .
      .
      .
```

It is clear from this example that a way to define groups of user processes (rather than giving several individual access identifiers) is needed.

Groups of user processes can be identified by a process class identifier, which, like the access identifier consists of a three-component name: Person_id.Project_id.tag. Although it can be identical to an access identifier, a process identifier generally is used to designate groups of processes by replacing one or more components with an asterisk (*). Since the asterisk means that any string can be in that component position, a process identifier with an asterisk identifies that group of processes whose access identifiers match the specifically named components (i.e., those components that are not an asterisk). For example, rather than the 12 entries needed in the above example, the user could specify all members of the Demo project, whether using interactive or absentee processes, by using one entry:

```
*.Demo.*   re
```

In addition, this single entry has the advantage of automatically providing for future members of the Demo project, because any access identifier matching the named component ("Demo") will have read and execute access to the segment.

Matching Entries on an ACL

A single process can be a member of more than one process class. This situation can lead to ambiguities on ACLs when more than one entry applies to the same process. To eliminate this ambiguity and make ACLs more easily readable, four conventions are imposed on ACLs and their interpretation. First, no process class identifier can appear more than once on any ACL. Second, the ACL is ordered as explained below. Third, the entry that applies to a given process is the first entry on the ordered list whose process class contains the given process. Finally, if no entry exists on the list for a given process, that process has no access to the segment or directory. These conventions ensure that the access for every process is uniquely specified by the ACL.

To properly generate and modify ACLs, it is necessary to have some understanding of how they are ordered. The ordering rule is to distinguish between specifically named components and asterisks, beginning with the leftmost component of the process identifier. The entries to be ordered are first divided into two groups, those whose first (Person_id) component is specific (i.e., not an asterisk) and those whose first component is an asterisk. Those with a specific first component are placed first on the ACL. Within these two groups, a similar ordering is done by second (Project_id) component with the specific entries again being first. This produces four groups. Finally, within each of these four groups, a similar ordering is done on the third (instance tag) component to produce eight groups.

The eight groups of class identifiers are ordered in the following manner:

1. no asterisks
2. an asterisk in the third component only
3. an asterisk in the second component only
4. asterisks in the second and third components only
5. an asterisk in the first component only
6. asterisks in the first and third components only
7. asterisks in the first and second components only
8. all asterisks (*.*.*)

Within each of these groups, the ordering is unimportant because a process can belong to only one class in a group. The following is a validly ordered ACL:

Jones.Work.a	r	(1)
Smith.Lazy.*	rw	(2)
White.*.a	re	(3)
Black.*.*	rew	(4)
*.Faculty.m	null	(5)
.Student.	re	(6)
.Lazy.	r	(7)
..z	rew	(8)
..*	r	(9)

In the above example, a process with access identifier Smith.Lazy.a would be able to read and write the segment as derived from entry (2), a process with access identifier Jones.Lazy.a would be able only to read the segment as derived from entry (7), and a process with access identifier Smith.Faculty.a would be able to read the segment as derived from entry (9). Despite entry (9), which apparently grants read access to all processes, Smith.Faculty.m would have no access since entry (5), which is encountered first, contains a more specific match (Faculty.m).

Maintenance of ACLs

Users can create and modify ACLs by using standard Multics commands and subroutines. The specific usage of each of these procedures is described in either the MPM Commands or MPM Subroutines:

list_acl
set_acl
delete_acl
hcs_$add_acl_entries
hcs_$add_dir_acl_entries

```
hcs_$replace_acl
hcs_$replace_dir_acl
hcs_$delete_acl_entries
hcs_$delete_dir_acl_entries
hcs_$list_acl
hcs_$list_dir_acl
```

The commands and subroutines enforce the constraints mentioned above; i.e., they order the ACL and do not permit more than one entry with a given process class identifier to appear on the ACL.

Consider the example of a segment with an ACL containing the single entry:

```
Jones.*.*     r
```

A new entry is added for the process class *.Work.* resulting in the ACL:

```
Jones.*.*     r

*.Work.*     rw
```

This would appear to give all members of the Work project the right to read and write the segment. Actually, it gives read and write access to all members of the Work project except Jones (assuming Jones is a member of the Work project). Jones has only read access. If the user truly wants to give all members of the Work project write access, he would have to either delete the Jones.*.* entry or add another entry to produce:

```
Jones.Work.*    rw

Jones.*.*        r

*.Work.*        rw
```

By keeping the Jones.*.* entry, the user continues to specify access for Jones when he logs in on any project other than Work.

It is important to realize that placing a new entry on an ACL does not necessarily grant all members of that process class the specified access, for some members of that process class can also be members of process classes appearing elsewhere on the ACL. The user should, therefore, be aware of what an ACL currently contains before modifying it.

Special Entries on an ACL

Several Multics system services are performed by special processes as opposed to being done by the user's process. These service processes 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 processes must have access to the segments to be serviced. These service processes, and only these service processes, are members of a single project called SysDaemon.

In order to ensure that these service processes have access to the segments, the storage system subroutines automatically place the ACL entry:

```
*.SysDaemon.*   rw
```

on the ACL of every segment, and the ACL entry:

```
*.SysDaemon.*   sma
```

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 service processes.

Under special circumstances, some user might not wish to use the facilities of a service process on some of his segments. In this case, the user simply denies that service process access to his segments by modifying the ACL entry (i.e., giving that service process null access). It is crucial that a user who elects not to receive such a system service be fully aware of the nature of the service and the consequences of his choice. For example, if the backup processes 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.

Initial ACLs

In addition to automatically adding a service process entry to the ACLs of all newly created segments and directories, many system commands and subroutines, e.g., create, create_dir, and hcs_$append_branch, add an entry for the creating process to the ACL of a newly created segment or directory. For convenience, the system allows a user to specify a list of entries to be added to all newly created segments or directories -- in addition to entries for the creating process and the service processes. This ability eliminates the need to explicitly modify an ACL each time a new segment or directory is created.

This list of ACL entries to be added to newly created segments or directories is called an initial access control list or initial ACL and is an attribute of a directory. Each directory has two sets of initial ACLs, one set for segments appended to the directory and one set for directories appended to the directory. Since each initial ACL is simply a list of ACL entries, an initial ACL has the appearance of an ACL. When a segment or directory is created, the service process ACL entry is first placed on the ACL of the segment or directory. Then, the appropriate initial ACL (i.e., either the one for segments or the one for directories) of the containing directory is merged with the ACL. The merging of two ACLs means that the entries are combined and sorted. If two entries on the resulting ACL contain the same process class identifier, the entry that was originally on the ACL of the segment is deleted, leaving the newly added entry. In this way, the service process entry originally on the segment can be overridden by placing an entry with process class identifier *.SysDaemon.* on the initial ACL. Finally, any entries specified in the operation of creating the segment or directory (for most system commands this is simply one entry for the creating process) are merged into the ACL. These entries override the service process and initial ACL entries if duplicate process class identifiers exist.

The default value for the initial ACLs of a newly created directory is empty, i.e., there are no entries in the initial ACLs.

Maintenance of Initial ACLs

The user can manipulate initial ACLs for either segments or directories using a set of commands and subroutines. The specific usage of each of these procedures is described in either the MPM Commands or the MPM Subsystem Writers' Guide:

```
delete_iacl_dir
delete_iacl_seg
list_iacl_dir
list_iacl_seg
set_iacl_dir
set_iacl_seg
hcs_$add_dir_inacl_entries
hcs_$add_inacl_entries
hcs_$delete_dir_inacl_entries
hcs_$delete_inacl_entries
hcs_$list_dir_inacl
hcs_$list_inacl
hcs_$replace_dir_inacl
hcs_$replace_inacl
```

NONDISCRETIONARY ACCESS CONTROL

Nondiscretionary access controls allow those responsible for administration of the organization using Multics to set up and enforce administrative policies on information access. When dealing with information of a sensitive nature, whether in the form of data bases or programs, an administrative policy is often necessary to control which persons may access certain kinds of information in specified ways. Such a policy could be implemented externally to Multics by requesting users to be careful about setting their access control lists. However, programming errors, oversights, and deliberate tampering can lead to the release or destruction of sensitive information. Multics enforces administrative policies through the use of the access isolation mechanism (AIM). Changes to the policies can be made only through administrators recognized by the system (e.g., system, system security, and project administrators).

As an example, consider a service bureau that serves several competitive customers. Each customer wishes to maintain and use sensitive information on the system, and wishes its employees to have access to it. Each customer wishes to prevent its competitors from accessing any of its sensitive information, as a result of accidents, carelessness, or by either the customer's or the service bureau's own employees deliberately "giving away" access to it. Further, the competitors would like to use the system's and each others' public programs on their sensitive data without fear of data disclosure or destruction. The nondiscretionary access control mechanism can be used to implement this policy.

Another example is a corporation that wishes its major divisions to use a common Multics system. The operating divisions should retain their privacy and independence, but corporate management requires some access to each operating division's financial and marketing data for reporting and planning. The operating divisions should not be able to access each other's data. The nondiscretionary access controls can be used to provide the desired blend of information sharing and isolation.

Some installations may not want to use nondiscretionary access controls. If the administrators feel that their organization does not need the service provided by AIM, they do not have to "do" anything to their Multics system. The default value of system_low puts all data at the lowest sensitivity level, which effectively cancels AIM checking and makes this type of access control invisible to all users.

AIM Attributes

Every segment and directory in the Multics storage hierarchy has associated with it -- for its lifetime -- an access class denoting the sensitivity of its contents. Every process (the active agent of a user) in the system has associated with it -- for its lifetime -- an access authorization denoting the sensitivity range of information it can access.

Because authorizations and access classes are identical in structure, the terms are often used interchangeably. An authorization or access class consists of a sensitivity level and a category set.

A sensitivity level is a single number conveying a relative sensitivity judgment. 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. Similarly, if process A is "more trusted" than process B, process A should have a higher sensitivity level than process B. The assignment of sensitivity levels is discussed later in this section.

Sensitivity levels are totally ordered. A process at one sensitivity level may access all segments and directories accessible to a process with a lower sensitivity level, subject to the other access rules (ACL and ring restrictions).

A category set is made up of zero or more access categories. An access category represents a grouping of information for access purposes. To illustrate how such groupings might be chosen, consider again the service bureau and corporation examples mentioned earlier. The service bureau might assign a different access category to each of its customers. This means that segments created by one customer will have a different access category than segments created by another customer. In the case of the corporation, a different access category might be assigned to each division. In this way, the different customers or the different divisions would all be isolated from one another (see "Relationships Between AIM Attributes" below for the meaning of isolated).

Category set access rules are based upon set inclusion. A process having a given category set may access all segments having the same category set or a subset of access categories, subject to ACL and ring restrictions.

If the category set of a process consists of several categories, this process has the need to access (and is trusted to access) several separate information groups. If the category set of a segment or directory consists of several categories, only processes that can access all of these several information groups will be allowed to access the contents of this segment or directory.

For convenience, the sensitivity levels and access categories used in a particular Multics installation are assigned character-string names by the system administration. 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."

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. Isolation is possible only if A and B have disjoint category sets, i.e., neither category set is equal to or a subset of the other. (An empty category set is considered a subset of all nonempty category sets.)

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 process may have read (r) and execute (e) modes to a segment only if the process authorization is greater than or equal to the segment access class.

2. A process may have write (w) mode to a segment only if the process authorization is equal to the segment access class.

3. A process has null access to a segment if its authorization is neither greater than nor equal to the segment access class.

DIRECTORIES

The rules for accessing directories are:

1. A process may have status (s) mode to a directory only if the process authorization is greater than or equal to the directory access class.

2. A process may have modify (m) and append (a) modes to a directory only if the process authorization is equal to the directory access class.

3. A process 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.

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.

MESSAGE SEGMENT

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.

Authorizations

Several authorization parameters are kept for users and projects in system and project tables.

For each person registered on the system, a person maximum authorization is kept (by the system security administrator) in a system table. This maximum authorization is composed of the highest sensitivity level and all the categories of information this person may ever access, on any project. Each installation has its own procedures for assigning and changing person maximum authorizations.

Each person registered on the system also has a default login authorization, kept in a system table, and changeable by the person himself (see login in the MPM Commands). If a person does not specify an authorization at log-in time, the default authorization is assumed.

For each project on the system, a project maximum authorization is kept (by the system security administrator) in a system table. This maximum authorization is composed of the highest sensitivity level and all the categories of information that any user logged in on this project may ever access. Again, each installation has its own procedures for assigning and changing project maximum authorizations.

For each person on a project (i.e., for each user), a user maximum authorization is kept (by the project administrator) in a project table. This maximum authorization is composed of the highest sensitivity level and all the categories of information this person may access when logged in on this project.

At the time a process is created (as the result of login or new_proc, both described in the MPM Commands), its authorization and maximum authorization are established.

The process maximum authorization is the highest authorization the process can attain. It is computed directly from the three maximum authorization parameters:

person maximum authorization

project maximum authorization

user maximum authorization on the project

The maximum authorization has the same form as an authorization or an access class. The sensitivity level of the process maximum authorization is the smallest of the sensitivity levels of the three maximums. The category set of the process maximum authorization is composed only of categories contained in the category sets of all three maximums. Thus, the process maximum authorization is the highest authorization that is less than or equal to each of the three maximum authorization parameters.

The process authorization is computed directly from the following parameters:

process maximum authorization

terminal access class

-auth argument to login or new_proc (or default)

The sensitivity level of the process authorization is computed in the same manner as that of the process maximum authorization -- it is the smallest of the sensitivity levels of the three parameters and its category set is composed only of categories contained in the category sets of the three parameters. Therefore, the process authorization is the highest authorization that is less than or equal to each of the three parameters. If the process authorization is not less than or equal to the process maximum authorization, then the process is not created, login or new_proc fails, and the system prints a message.

Access Classes

The access classes assigned by AIM to segments, directories, and message segments are described below.

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. All upgraded directories must have a terminal quota. 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 "nondecreasing access class" rule.

An upgraded directory must have a terminal 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).

MESSAGE SEGMENT

As explained earlier, 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 process 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 process that created it. The access class of the message segment must be greater than or equal to the access class of its containing directory.

Maintenance of AIM

Standard system commands and subroutines allow users to make use of nondiscretionary access controls. In some cases, administrative personnel are needed to perform changes or maintenance activities. Also, the use of nondiscretionary access control imposes some restrictions that the user should be aware of. Maintenance activities performed by both users and administrators as well as restrictions under AIM are described below.

USER COMMANDS AND SUBROUTINES

There are several commands and subroutines that deal explicitly with authorizations and access classes. Each one is described in detail in either the MPM Commands or the MPM Subroutines:

```
commands:
     create_dir
     login
     new_proc
     print_auth_names
     print_proc_auth
     status

subroutines:
     convert_authorization_
     get_authorization_
     get_max_authorization_
     hcs_$create_branch_
     hcs_$get_access_class_seg
```

SPECIAL SITUATIONS

It is sometimes necessary to lower the access class of some information (i.e., downgrade it). The nondiscretionary access control rules do not allow user processes to perform this type of operation. It is necessary for a system security administrator, whose process has special accessing privileges, to make the change. Each installation has its own procedures for users to request such changes.

Occasionally, through system failure, inconsistencies in the access classes of directories and segments may develop. Examples of such inconsistencies are a directory with an access class that is not greater than or equal to that of its containing directory; or a segment with an access class not equal to that of its containing directory. Such a situation is reported with the system status code error_table_$oosw, or the error message "There was an attempt to reference a directory which is out of service." (See Section VII for a list of error messages.) In this case, it is necessary to have a system security administrator examine and correct the inconsistencies, and to place the directories back in service. Each installation has its own procedures for reporting such problems.

MAILBOXES

The mail command (described in the MPM Commands) uses message segments to hold mail, each piece of mail being a single message with an access class equal to the authorization of the sending process. Nondiscretionary access controls impose several restrictions on the use of mail. Since a process can read only messages with access classes less than or equal to its authorization, a process cannot read mail sent by processes of higher authorizations. Since a process can only delete messages with access classes equal to its authorization, it cannot delete mail with an access class not equal to its authorization.

The access class of a mailbox is equal to the maximum authorization of the process 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 process 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 processes at system_low 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.

GENERAL RESTRICTIONS

AIM imposes restrictions on the use of several system facilities. The nondiscretionary access controls allow a user to log in a process at different authorizations. Since AIM enforces access rules based on these authorizations, a user's access to certain objects may differ at different logins.

Any operation performed by the user that requires write access to a given segment (or modify or append access to a given directory) can only succeed if the process authorization is equal to the access class of the segment (or directory). Likewise, any operation performed by the user that requires read access to a given segment (or status access to a given directory) can only succeed if the process authorization is greater than or equal to the access class of the segment (or directory). For example, if the user creates his profile segment at his maximum authorization, he will be unable to use it (read it) at lower authorizations. Therefore, for convenience, most users choose to create their profile segments at system_low. However, in this case, the user cannot add or delete abbreviations at any authorizations higher than system_low. Similar restrictions apply to data segments used by the memo, debug, and probe commands. Also, general purpose exec_com segments that modify a segment or directory can only be used when the process authorization equals the segment or directory access class.

To summarize, the full use of many system facilities is inherently restricted to a single authorization. Persons who choose to work at more than one authorization may find these system facilities unusable or only partially usable at certain authorizations.

INTRAPROCESS ACCESS CONTROL

The ring mechanism permits users to write subsystems that are protected from other users in much the same way the supervisor is protected from users. For example, consider a user who has created a segment containing a list of his employees and their salaries. He wants to permit some users to have access to the list of employees and allow some other users to know the average salary of his employees, but he does not want anyone but himself to have access to the specific salary data for each employee. In other words, he wishes other users to have access to the data in the segment but only in a controlled manner and he wants to be able to specify the control. The read, write, and execute access modes of segments do not provide this capability; however, the ring mechanism does. In effect, rings permit arbitrarily refined and controlled access to objects by allowing any user to define arbitrary objects, write procedures that operate on these objects, and encapsulate these procedures and objects in a closed, controlled environment that can be entered only at specific entry points.

One of the important properties of the ring mechanism is that it is invisible to users of segments protected by it. Only those programmers actually writing subsystems that need ring protection need be familiar with the use and operation of rings.

Conceptually, the Multics ring mechanism could be pictured as a series of concentric circles. All the segments in a process reside somewhere within these circles; that is, each segment is located either between the boundaries of some pair of circles, or in the innermost circle. These circles (or rings) are numbered from 0 to 7, starting with the innermost circle, to denote levels of privilege; ring 0 is the ring of most privilege and ring 7 is the ring of least privilege. A process executing in a given ring has free access to other processes executing in that same ring, subject to any limitations prescribed by the other access restrictions (ACL and AIM). Access between rings is limited according to rules given below.

The primary rule of access between rings is that processes executing in lower-numbered rings have unlimited access to segments in higher-numbered rings, subject, of course, to ACL and AIM restrictions on particular segments. Processes executing in higher-numbered rings have no direct access to segments in lower-numbered rings. Access refers to both the ability to execute a segment and the ability to read or write it. Thus the ring boundaries are _walls_. Within a ring (i.e., between walls) processes are unimpeded by the ring protection mechanism. It is when a wall must be crossed that the protection mechanism becomes effective.

Those segments that compose the central supervisor are in ring 0. Ring 1 usually contains system routines, largely administrative in nature, that are not as sensitive as the central supervisor. Rings 3 through 7 are potentially available for use by users. Most user processes start running in ring 4. Rings 5 through 7 are available for use by programmers who wish to write their own subsystems.

In a process, the ring that contains the currently executing segment is called the current ring of execution and is part of the state of the process. Control must, of course, be able to pass from ring to ring. By virtue of the ring structure's basic definitions, passing control outward is legal. That is, segments in outer rings are accessible to those in inner rings. However, by virtue of those same definitions, an inward call cannot be legal. Therefore, segments in inner rings are inaccessible to those in outer rings. The means of legitimizing inward calls is to cause one or more entry points of a given procedure segment to be treated as gates in the protection wall. A gate, then, is an entry point to an inner ring procedure segment that can be called by an outer ring segment.

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.

Segment Ring Brackets

Thus far, it has been indicated that each segment in the system must be a member of a single ring and, if the segment is executable, execute in that ring. It is, however, convenient to allow a segment to reside in (be a member of) several rings so that it can execute in any of these rings with the access appropriate to that ring. This is accomplished by giving each segment an execute bracket that delimits the rings in which the segment can be executed (if it has execute access mode), without having to change the ring of execution of the process. The execute bracket is specified by means of two ring numbers, for example, 3, 5. The execute bracket includes all rings between and including the two ring numbers; in this case, the execute bracket contains rings 3, 4, and 5.

If the process is executing in a ring contained in the execute bracket of a segment and control is transferred to the segment, then no change of ring of execution results. If the process is executing in a ring whose number is less than the lowest ring number in the execute bracket, then when the segment is transferred to, the ring of execution will be changed to the lowest ring in the execute bracket. In the example above, if the process is executing in ring 1 and the segment is transferred to, then the ring of execution will become 3. If the process is executing in a ring whose number is greater than the highest ring number in the execute bracket, then the ring of execution will become the highest ring in the execute bracket (5 in the example) when the segment is transferred to, assuming the segment contains a gate. (As stated earlier, a gate is an entry point to an inner ring procedure segment that can be called by an outer ring segment.)

In this latter case of gates, it is also useful to specify those rings in which the segment is accessible through a gate. This gate bracket is specified by appending a third ring bracket number after the two already used for defining the execute bracket, e.g., 3,5,6. The gate bracket includes those rings whose number is greater than the second ring bracket number and less than or equal to the third bracket number (in the example, only ring 6). An attempt to execute a segment from a ring greater than the gate bracket is not allowed.

Since a segment must have a nonempty gate bracket in order to contain a gate, it is convenient to choose a nonempty gate bracket for a segment as the definition of a gate segment; e.g., an executable segment with ring bracket numbers 4,5,7 is a gate segment because the gate bracket contains rings 6 and 7, whereas an executable segment with ring bracket numbers 4,5,5 is not a gate segment because its gate bracket is empty. Gate segments must also have a specific format. The above use of ring bracket numbers dictates that they be increasing; i.e., the first ring bracket number must be less than or equal to the second ring bracket number which, in turn, must be less than or equal to the third ring bracket number.

The ring bracket numbers also have meaning with respect to the read and write access modes. The rings less than or equal to the first of the ring bracket numbers are termed the write bracket. A process 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 in the segment. If a process is running in a ring higher than the write bracket, it cannot modify (write into) the segment even though the process has write access mode on the segment. The rings less than or equal to the second ring bracket number are called the read bracket. Processes must be running in the read bracket of a segment and have read access mode on the segment in order to read it. The list below summarizes the refinements of access that are controlled by the ring brackets of a segment and the process' ring of execution, assuming the process has read, write, and execute access modes specified on the ACL of the segment. Ring brackets do not grant access to a segment; access to a segment is granted only by the ACL and AIM controls. Ring brackets only serve to refine, within the process, the access granted by the ACL and AIM controls.

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

Directory Ring Brackets

Directory ring brackets are in most ways similar to segment ring brackets, but with two important differences:

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 process to modify or add entries to a directory, the validation level of the process must be within the modify/append bracket and the process 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 process can get status of, modify, or append to the directory (assuming, of course, that it has the status, modify, and append access modes). If the validation level is 6, it can only get status of the directory. If the validation level is 7, it cannot access the attributes of the entries in the directory at all.

The ring brackets of segments or directories can be modified by using a set of commands or subroutines described in the MPM Subsystem Writers' Guide:

```
set_ring_brackets
hcs_$set_ring_brackets
hcs_$set_dir_ring_brackets
```

Modification of Segment Attributes

In order to maintain the integrity of the ring mechanism, the ring brackets of a segment or directory must control the ability of a process to modify the attributes (particularly the ACL and ring brackets) of a segment as well as the ability to write the data of the segment. As stated previously, in order to modify the attributes of a segment in a directory, or of an inferior directory, the process must have modify access to the latter directory and the validation level must be within the modify/append bracket of the directory. A further qualification is that the validation level must be within the write bracket of a segment whose attributes are being modified or be within the modify bracket of a directory whose attributes are being modified. Also, a process cannot set bracket numbers to values less than the validation level. Finally, to prevent one protected subsystem from tampering with another protected subsystem in the same ring, an ACL entry containing a project identifier other than the project of the executing process cannot be added to the ACL of a gate segment. These restrictions ensure that there are no means by which a process executing in a ring outside the write bracket can directly write a segment or do so indirectly by first modifying the ring brackets or ACL of the segment to give the process write access and then write it.

The final type of directory entry, the link, has no access control list or ring brackets of its own. To modify or delete a link, the process must have modify access mode on the directory containing the link, and the validation level must be within the modify bracket of the directory.

Default Values

When a segment or directory is created and the values for the ring bracket numbers are not explicitly defined, they will be set to a default value equal to the validation level. Since most users write programs that operate in a single ring at a single validation level, this choice of default ring brackets makes the ring mechanism invisible to them, i.e., they will always be within the read, write, and execute brackets for segments and the status and modify/append brackets for directories.

As explained in "Dicretionary Access Control" above, when a new segment or directory is appended to a directory, the default value for the access control list is determined by the initial ACLs. In each directory, there is a set of initial ACLs for newly created segments and a set of initial ACLs for newly created directories. The reason why a set of initial ACLs rather than simply a single initial ACL is necessary is that the set contains one initial ACL for each ring. When a segment or directory is created, the initial ACL corresponding to the validation level is the one used. Since the initial ACL for a given ring can be modified only by procedures in rings equal to or less than the given ring, a procedure creating a new segment or directory can be sure that the initial ACL to be used could not have been modified by a ring less privileged than the ring on whose behalf the segment or directory is being created.

The "user ring" is ring 4, because most user processes start running in ring 4; however, rings 3 through 7 are also available for user processes. The project administrator may specify both the initial ring for a user process and a limit on the maximum ring that may be used. A ring attribute of 4,6 would start the user's process in ring 4 and allow it to execute also in rings 5 and 6.

SECTION VII

HANDLING UNUSUAL OCCURRENCES

A procedure may encounter a set of circumstances that prevent it from continuing normally. Examples of circumstances that prevent a procedure 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 procedure from continuing in a normal manner is dependent upon the procedure in question. Circumstances that are abnormal for one procedure can be quite normal when encountered in a different procedure. If a procedure 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 procedures. However, it should help users select appropriate means for handling and reporting unusual occurrences that arise during the execution of their own procedures. 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 MPM Subroutines 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 either a short bit string or arithmetic number that takes on a different value for each possible unusual occurrence. If the status code is a bit string, usually each bit refers to the occurrence of some circumstance, as in Multics I/O system status codes. If the status code is an arithmetic number, then each different value corresponds to an unusual occurrence or set of unusual occurrences, as in the case of the Multics storage system status codes. 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.

Multics provides a means by which status codes can be generated and interpreted. The status codes generated are one word arithmetic numbers (fixed binary(35)) whose scope is a single process. The actual values of the codes are generated dynamically when referenced symbolically from a program, and can be interpreted (i.e., converted to a character-string description) by calling the com_err_ subroutine. By using these dynamically generated status codes rather than status codes with fixed, preassigned values, conflict is avoided between several separately compiled subsystems that can all use the same status code to represent different occurrences. In the dynamic scheme, all status codes are guaranteed to be unique within a process. Status codes cannot be used in a process other than the one generating them because they do not necessarily have the same interpretation in another process.

In order to have a status code generated, a Multics standard status code segment must exist. (A description of how to generate a standard status code segment is given in the error_table_compiler command description in the MPM Subsystem Writers' Guide.) 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 new value for each status code defined in the segment and stores it in the segment. (Actually, it is stored in the linkage section of the status code segment, so that a different status code can be generated for each process.) 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.

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. Several of the status codes listed below pertain directly to nondiscretionary access control. (See "Nondiscretionary Access Control" and "Special Situations" in Section VI for information on the access isolation mechanism and errors that can occur.)

To test for the return of a particular system-defined status code, the following approach can be taken in order to avoid compiling particular numeric values, which might change, into programs:

```
     declare error_table_$entry

     if code = error_table_$entry then ...
```

where:

1. code — is a status code (fixed bin(35)) returned from a Multics system command or subroutine.

2. entry — is an error_table_ entry taken from the list below.

Storage System Status Codes

a_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_string
    Unable to convert access class/authorization to binary.

ai_restricted
    Improper access class/authorization to perform operation.

argerr
    There is an inconsistency in arguments to the storage system.

bad_acl_mode
Bad mode specification for ACL.

bad_name
The access name specified has an illegal syntax.

bad_ring_brackets
Ring brackets input to directory control are invalid.

boundviol
Attempt to access beyond end of segment.

clnzero
There was an attempt to move segment to non-zero length entry.

dirseg
This operation is not allowed for a directory.

empty_acl
ACL is empty.

full_hashtbl
The directory hash table is full.

fulldir
There was an attempt to delete a non-empty directory.

incorrect_access
Incorrect access to directory containing entry.

infcnt_non_zero
There was an attempt to make a directory unknown that has inferior segments.

invalid_ascii
The name specified contains non-ascii characters.

invalid_copy
There was an attempt to create a copy without correct access.

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_quota
Invalid move of quota would change terminal quota to nonterminal.

invalid_project_for_gate
Invalid project for gate access control list.

invalidsegno
There was an attempt to use an invalid segment number.

max_depth_exceeded
The maximum depth in the storage system hierarchy has been exceeded.

moderr
Incorrect access on entry.

name_not_found
The name was not found.

namedup
Name duplication.

no_dir
Some directory in path specified does not exist.

no_info
Insufficient access to return any information.

no_makeknown
Unable to make original segment known.

no_move
Unable to move segment because of type, access or quota.

no_s_permission
Status permission missing on directory containing entry.

noalloc
There is no room to make requested allocations.

noentry
Entry not found.

nomatch
Use of star convention resulted in no match.

nonamerr
The operation would leave no names on entry.

nondirseg
This operation is not allowed for a segment.

not_a_branch
Entry is not a branch.

notadir
Entry is not a directory.

notalloc
     Allocation could not be performed.
nrmkst
     There is no more room in the KST.
oldnamerr
     Name not found.
refname_count_too_big
     The reference name count is greater than the number of reference names.
root
     The directory is the ROOT.
rqover
     Record quota overflow.
safety_sw_on
     Attempt to delete segment whose safety switch is on.
seg_unknown
     Segment not known to process.
segknown
     Segment already known to process.
segnamedup
     Name already on entry.
too_many_sr
     Too many search rules.
toomanylinks
     There are too many links to get to a branch.
user_not_found
     User name not on access control list for branch.

Input/Output System Status Codes

already_assigned
     Indicated device assigned to another process.
att_loop
     Attachment loop.
bad_arg
     Illegal command or subroutine argument.
bad_file
     File is not a structured file or is inconsistent.
bad_label
     Incorrect detachable medium label.
bad_mode
     Improper mode specification for this device.
bad_mount_request
     Mount request could not be honored.
bad_tapeid
     Invalid volume identifier.
bad_volid
     Invalid volume identifier.
blank_tape
     The rest of the tape is blank.
buffer_big
     Specified buffer size too large.
cyclic_syn
     Cyclic synonyms.
data_improperly_terminated
     Relevant data terminated improperly.
dev_nt_assnd
     IO device not currently assigned.
dev_offset_out_of_bounds
     Specified offset out of bounds for this device.
device_end
     Physical end of device encountered.
device_limit_exceeded
     The process's limit for this device type is exceeded.
device_parity
     Unrecoverable data-transmission error on physical device.

end_of_info
End of information reached.

file_busy
File already busy for other I/O activity.

incompatible_attach
Attach and open are incompatible.

insufficient_open
Insufficient information to open file.

invalid_backspace_read
Invalid backspace_read order call.

invalid_device
Attempt to attach to an invalid device.

invalid_elsize
Invalid element size.

invalid_read
Attempt to read or move read pointer on device which was not attached as readable.

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_state
Request is inconsistent with current state of device.

invalid_write
Attempt to write or move write pointer on device which was not attached as writeable.

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.

key_order
Key out of order.

long_record
Record is too long.

missent
Missing entry in outer module.

mount_not_ready
Requested volume is not yet mounted.

mount_pending
Mount request pending.

multiple_io_attachment
The stream is attached to more than one device.

negative_nelem
Negative number of elements supplied to data transmission entry.

negative_offset
Negative offset supplied to data transmission entry.

no_backspace
Requested tape backspace unsuccessful.

no_device
No device currently available for attachment.

no_io_interrupt
No interrupt was received on the designated IO channel.

no_iocb
No I/O switch.

no_key
No key given for write_record.

no_operation
Invalid I/O operation.

no_record
    Record not located.
no_wired_structure
    No wired structure could be allocated for this device request.
not_attached
    Process not attached to indicated device.
not_closed
    I/O switch is not closed.
not_detached
    I/O switch is not detached.
not_open
    I/O switch is not open.
old_dim
    Old DIM cannot accept new I/O call.
redundant_mount
    Requested volume is already mounted.
short_record
    Record is too short.
tape_error
    Tape error.
too_many_buffers
    Too many buffers specified.
too_many_read_delimiters
    Too many read delimiters specified.
typename_not_found
    Typename not found.
unable_to_do_io
    Unable to perform critical I/O.
undefined_order_request
    Undefined order request.
unregistered_volume
    The specified detachable volume has not been registered.
wakeup_denied
    Attempt to wakeup a process of lower authorization.

Other Status Codes

action_not_performed
    The requested action was not performed.
arg_ignored
    Argument ignored.
bad_arg_acc
    Improper access to given argument.
bad_arg_type
    Bad gate for entry referenced.
bad_class_def
    Bad class code in definition.
bad_command_name
    Improper syntax in command name.
bad_entry_point_name
    Illegal entry point name in make_ptr call.
bad_equal_name
    The equal name specified had illegal syntax.
bad_index
    Internal index out of bounds.
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_ms_convert
    Attempt to convert directory or link to multisegment file.
bad_ms_file
    Directory or link found in multisegment file.
bad_processid
    Current processid does not match stored value.

bad_ptr
Argument is not an ITS pointer.

bad_segment
There is an internal inconsistency in the segment.

bad_self_ref
Illegal self reference type.

bad_string
Unable to process a search rule string.

badcall
Procedure called improperly.

badequal
Illegal use of equals convention.

badopt
Specified control argument is not implemented by this command.

badpath
Bad syntax in pathname.

badstar
Illegal entry name.

badsyntax
Syntax error in ascii segment.

bigarg
Argument too long.

command_line_overflow
Expanded command line is too large.

date_conversion_error
Unable to convert character date/time to binary.

defs_loop
Looping searching definitions.

dirlong
Directory pathname too long.

dup_ent_name
Duplicate entry name in bound segment.

ect_full
The event channel table was full.

entlong
Entry name too long.

force_bases
No bases supplied in force call.

id_already_exists
Supplied identifier already exists in data base.

improper_data_format
Data not in expected format.

inconsistent
Inconsistent combination of control arguments.

inconsistent_ect
The event channel table was in an inconsistent state.

invalid_channel
The event channel specified is not a valid channel.

invalid_lock_reset
The lock was locked by a process that no longer exists, therefore the lock was reset.

lesserr
Too many "<"'s in pathname.

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 that was locked by another process.

locked_by_this_process
The lock was already locked by this process.

longeql
Equals convention makes entry name too long.

loterr
Error zeroing entry in the linkage offset table.

mismatched_iter
Mismatched iteration sets.

no_defs
Bad definitions pointer in linkage.

no_ext_sym

External symbol not found.

no_linkage

Linkage section not found.

no_restart

Supplied machine conditions are not restartable.

no_sym_seg

Symbol segment not found.

noarg

Expected argument missing.

nodescr

Expected argument descriptor missing.

nolinkag

No/bad linkage info in the lot for this segment.

nolot

No linkage offset table in this ring.

nostars

The star convention is not implemented by this procedure.

not_act_fnc

Procedure was not invoked as an active function.

not_bound

Segment is not bound.

not_seg_type

Segment not of type specified.

odd_no_of_args

Odd number of arguments.

out_of_sequence

A call that must be in a sequence of calls was out of sequence.

pathlong

Pathname too long.

recursion_error

Infinite recursion.

request_not_recognized

Request not recognized.

sameseg

Attempt to specify the same segment as both old and new.

seg_not_found

Segment not found.

smallarg

Argument size too small.

stack_overflow

Not enough room in stack to complete processing.

strings_not_equal

Strings are not equal.

too_many_args

Maximum number of arguments for this command exceeded.

translation_aborted

Fatal error. Translation aborted.

translation_failed

Translation failed.

unbalanced_brackets

Brackets do not balance.

unbalanced_parentheses

Parentheses do not balance.

unbalanced_quotes

Quotes do not balance.

unimplemented_version

This procedure does not implement the requested version.

useless_restart

The same fault will occur again if restart is attempted.

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. This is the purpose of the Multics condition mechanism (described in "Multics Condition Mechanism" below.).

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 enviroment 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; an attempt by a user to exceed his storage allocation limit is a condition identified by the name, record_quota_overflow.

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 while 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.

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.

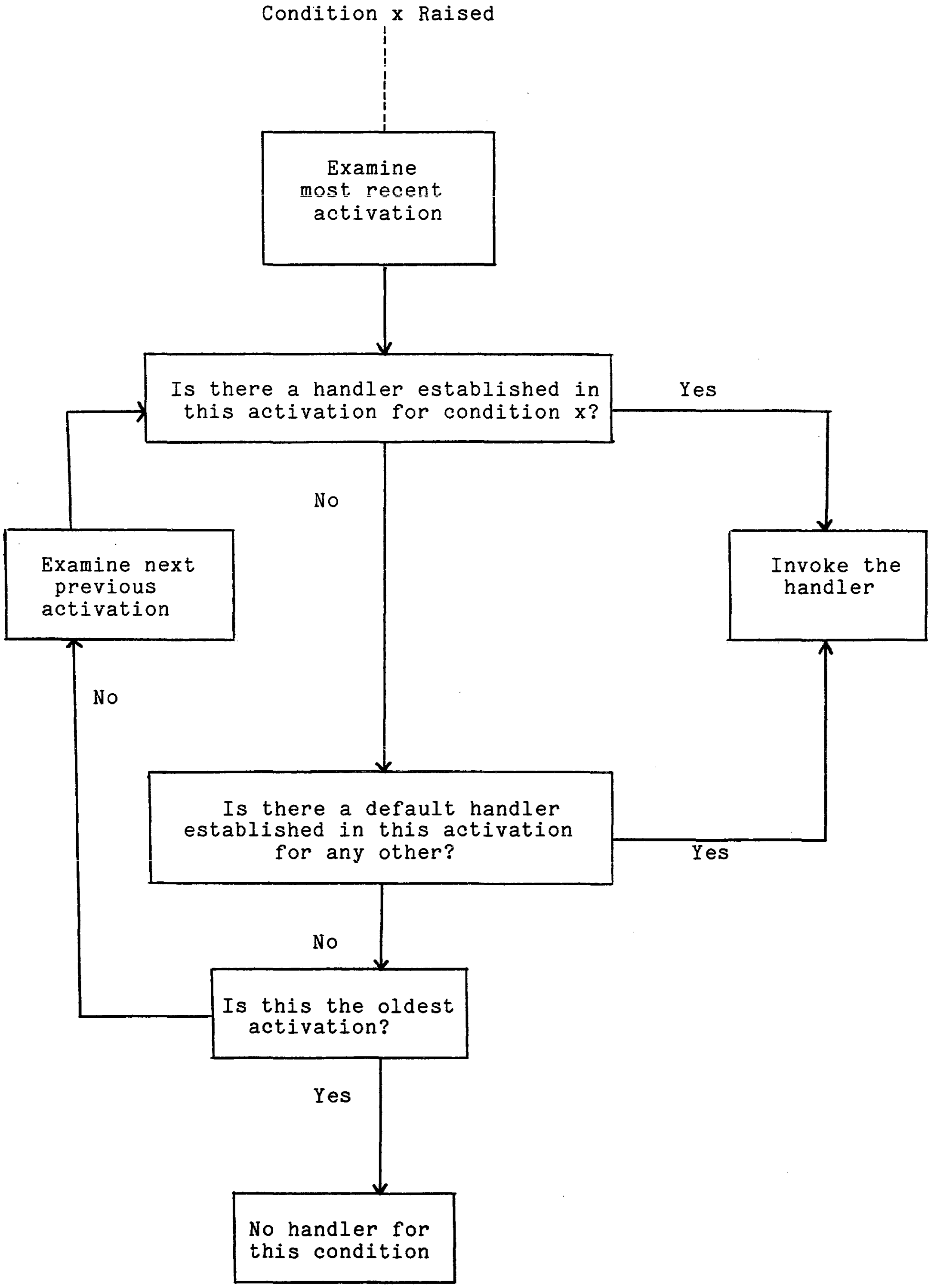

Condition x Raised
Examine most recent activation
Is there a handler established in this activation for condition x?
Yes
No
Examine next previous activation
Invoke the handler
No
Is there a default handler established in this activation for any other?
Yes
No
Is this the oldest activation?
Yes
No handler for this condition

Figure 7-1. Simplified Handler Algorithm.

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.

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.

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 MPM Subsystem Writers' Guide) 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 Section VII of the MPM Subsystem Writers' Guide) 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. It has the following declaration:

```
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 reserved1                bit(108),
      2 errcode fixed            bin(35),
      2 reserved2                bit(72),
      2 ring                     bit(18),
      2 fault_time               bit(54),
      2 reserved3 (0:7)          bit(36)) unaligned;
```

where:

1. prs — is the contents of the eight pointer registers at the time the condition occurred.

2. 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 accumulator contents.

   q — is the q-register contents.

   e — is the exponent register contents.

3. scu — is the stored control unit, expanded below.

4. errcode — is the fault error code. Refer to "List of System Status Codes and Meanings" earlier in this section.

5. ring — is the ring in which the condition occurred.

6. 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,

     /* WORD (0) */

    (2 ppr,
       3 prr            bit(3),
       3 psr            bit(15),
       3 p              bit(1),
     2 reserved4        bit(17),
/* WORD (1) */

2 reserved5       bit(35),
2 fi_flag         bit(1),

/* WORD (2) */

2 tpr,
  3 trr           bit(3),
  3 tsr           bit(15),
2 reserved6       bit(18),

/* WORD (3) */

2 reserved7       bit(30),
2 tpr_tbr         bit(6),

/* WORD (4) */

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),

/* WORD (5) */

2 ca              bit(18),
2 reserved8       bit(18),

/* WORD (6) */

2 even_inst       bit(36),

/* WORD (7) */

2 odd_inst        bit(36)) aligned;
```

where:

1. ppr is the procedure pointer register contents.

prr is the ring number portion of ppr.

psr is the segment number portion of ppr.

p is the procedure privileged bit.

2. fi_flag is the fault/interrupt flag.
"0"b interrupt
"1"b fault

3. tpr is the temporary pointer register contents.

trr is the ring number portion of tpr.

tsr is the segment number portion of tpr.

4. tpr_tbr is the bit offset portion of tpr.

5. ilc is the instruction counter contents.

6. 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 multiword instruction mode.

abs absolute mode.

7. ca is the computed address.

8. even_inst the instruction causing the fault is stored here.

9. 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 is particular to the condition in question and varies among conditions. The format of that header is:

```
dcl 1 info_structure              aligned,
      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 reserved                bit(34) unaligned,
      2 info_string               char(256) var,
      2 status_code               fixed bin(35);
```

where:

1. length — is the length of the structure in words.

2. version — is the version number of this structure.

3. 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 a return.

   reserved — is reserved for use by the system.

4. info_string — is a printable message about the condition.

5. status_code — if nonzero, is a code interpretable by the com_err_ subroutine that further defines the condition.

If neither action flag is set, restarting is possible, but its success depends on the action taken by the handler.

LIST OF SYSTEM CONDITIONS AND DEFAULT HANDLER

to report certain unusual occurrences encountered by system procedures. 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.

Four 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: indicate when the user should not attempt to handle the condition and note when restarting after an occurrence of the condition is inappropriate;

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.

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 ondata 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".

```
dcl 1 info                     aligned,
     2 length                     fixed bin,
     2 version                    fixed bin,
     2 action_flags               aligned,
       3 cant_restart             bit(1),
       3 default_restart          bit(1),
       3 reserved                 bit(34),
     2 info_string                char(256) var,
     2 status_code                fixed bin(35),
     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;
```

where:

1. - 4. is the same as in the information header format above.

5. status_code is the status code, if any, that caused the condition to be signalled.

6. id identifies this structure as belonging to a PL/I condition.

7. v1_sw indicates that the condition was raised by a version 1 PL/I procedure.
"1"b condition was raised
"0"b condition was not raised

8. oncode_sw indicates that the structure contains a valid oncode.
"1"b oncode valid
"0"b no valid oncode present

9. onfile_sw indicates that a file name has been copied into the structure.
"1"b name copied
"0"b name is not copied

10. file_ptr_sw indicates that there is a file associated with this condition.
"1"b file associated
"0"b file is not associated

11. 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

12. 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

13. 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

14. 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

15. oncode is the condition's oncode if oncode_sw is equal to "1"b.

16. onfile is the onfile string if onfile_sw is equal to "1"b.

17. file_ptr is a pointer to a file value if file_ptr_sw is equal to "1"b.

18. onsource is the onsource string if onsource_sw is equal to "1"b.

19. oncharindex is the character offset in onsource of the erroneous character if onchar_sw is equal to "1"b.

20. 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

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 hold the stack for further debugging activities or he can release it. (See the description of the debug, release, and start commands in MPM Commands.)

active_function_error

Cause: the user incorrectly used an active function in a command line. The active_fnc_err_ subroutine (described in the MPM Subsystem Writers' Guide) signals this condition.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure:

```
dcl 1 active_function_error_info  aligned,
     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 reserved                 bit(34) unaligned,
     2 info_string                char(256) var,
     2 status_code                fixed bin(35),
     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);
```

where:

1. - 4. is the same as in the information format header above.

5. status_code is the status code being reported by the active_fnc_err_ subroutine.

6. name_ptr is a pointer to a character string containing the name of the procedure that called the active_fnc_err_ subroutine.

7. name_lth is the length of the name of the procedure that called the active_fnc_err_ subroutine.

8. errmsg_ptr — is the significant length of the error message prepared by the active_fnc_err_ subroutine. A handler might wish to alter that message.

9. 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.

10. max_errmsg_lth — is the size of the character string containing the error message prepared by the active_fnc_err_ subroutine.

11. 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 user to the timer_manager_$alarm_call entry point (to set the alarm). See the description of the timer_manager_ subroutine in the MPM Subsystem Writers' Guide.

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.

Data structure: none.

Note: the any_other condition handlers should pass this on.

area (PL/I)

Cause: the user attempted to either allocate storage in an area that had insufficient space remaining to generate the storage needed; or assign one area to another, and the second had insufficient space to hold the storage allocated in the first. 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.

Data structure: none.

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.

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 MPM Subroutines) 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.

Data structure:

```
dcl 1 command_error_info        aligned,
      2 length                  fixed bin,
      2 version                 fixed bin init(2),
      2 action_flags            aligned,
        3 cant_restart          bit(1) unaligned,
        3 default_restart       bit(1) unaligned,
        3 reserved              bit(34) unaligned,
      2 info_string             char(256) var,
      2 status_code             fixed bin(35),
      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);
```

where:

1. - 4. is the same as in the information header format above.

5. status_code is the status code reported by the com_err_ subroutine.

6. name_ptr is a pointer to a character string containing the name of the procedure that called the com_err_ subroutine.

7. name_lth is the length of the name of the procedure that called the com_err_ subroutine.

8. 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.

9. errmess_lth — is the significant length of the error message prepared by the com_err_ subroutine. This datum can be changed by the handler.

10. max_errmess_lth — is the size of the character string containing the error message prepared by the the com_err_ subroutine.

11. 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 MPM Subroutines) signals this condition.

Default action: prints a message and returns to command level.

Restrictions: none.

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.

Data structure:

```
dcl 1 command_question_info,
      2 length                  fixed bin,
      2 version                 fixed bin init(2),
      2 action_flags            aligned,
        3 cant_restart          bit(1) unaligned,
        3 default_restart       bit(1) unaligned,
        3 reserved              bit(34) unaligned,
      2 info_string             char(256) var,
      2 status_code             fixed bin(35),
      2 reserved                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;
```

where:

1. - 4. is the same as in the information header format above.

5. status_code is the status code that prompted the call to the command_query_ subroutine.

6. reserved is currently ignored. (A value of zero is always passed to the handler.)

7. 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

8. 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 no answer required

9. 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

10. 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

11. name_ptr is a pointer to a character string containing the error message prepared by the command_query_ subroutine.

12. name_lth is the length of the name of the procedure that called the command_query_ subroutine.

13. 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.

14. question_lth is the significant length of the question pointed to by question_ptr. This datum can be changed by the handler.

15. max_question_lth is the size of the character string pointed to by question_ptr.

16. answer_ptr is a pointer to a character string that can be used by the handler to return a preset answer.

17. answer_lth is the significant length of the preset answer pointed to by answer_ptr. This datum can be changed by the handler.

18. max_answer_lth is the size of the character string pointed to by answer_ptr.

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.

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 MPM Subsystem Writers' Guide.)

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.

Data structure: none.

Note: the any_other condition handlers should pass this on.

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.

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.

Data Structure: none.

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.

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.

Data structure: the standard PL/I data structure.

Note: 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.

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: if the error condition is not merely an echo of another PL/I condition, then restarting (i.e., returning control to the signaller) is usually undefined. Restarting from other PL/I conditions is discussed under the individual conditions.

Data structure: the standard PL/I data structure.

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.

Data structure: none.

finish

Cause: the user's process is being terminated by a logout (either voluntary or involuntary) or by a new_proc command (described in the MPM Commands).

Default action: closes all open files and returns.

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.

Data structure: none.

Note: the any_other condition handlers should pass this on.

fixedoverflow (hardware)

Cause: the result of a binary fixed-point operation exceeded 71 bits, or the result of a decimal fixed-point operation exceeded 63 digits.

Default action: prints a message on the error_output switch and signals the error condition.

Restrictions: a return to the point where the signal occurred is prohibited since continued execution from this point is undefined.

Data structure: none.

gate_error

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.

Data structure: none.

illegal_modifier (hardware)

Cause: an invalid modifier appeared on an indirect word.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: in a machine language program this could be a simple programmer error, a compiler error, or a hardware error.

illegal_opcode (hardware)

Cause: the user attempted to execute an illegal operation code. In a machine language program this could be a simple programmer error. It could also be a compiler error or a hardware error.

Default action: prints a message and returns to command level.

Restrictions: none.

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.

Data structure: none.

Note: in a machine language program this could be a programmer error, a compiler error, or a hardware error.

illegal_return

Cause: an attempt was made to restart machine condition with invalid information.

Default action: prints a message and returns to command level.

Restrictions: none.

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 MPM Subroutines.)

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure:

```
dcl 1 io_error_info               aligned,
     2 length                    fixed bin,
     2 version                   fixed bin init (0),
     2 action_flags              aligned,
       3 cant_restart            bit(1) unaligned,
       3 default_restart         bit(1) unaligned,
       3 reserved                bit(34) unaligned,
     2 info_string               char(256) var,
     2 status_code               fixed bin(35),
     2 stream                    char(32),
     2 status                    bit(72);
```

where:

1. - 4. is the same as in the information header format above.

5. status_code is the unexpected status code received by an I/O procedure.

6. stream is the name of the switch on which the I/O operation was performed.

7. status is the I/O system status code describing the error.

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.

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.

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 or the entry point did not exist in that segment.

Default action: prints a message and returns to command level.

Restrictions: none.

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.

Data structure: none.

message_segment_error

Cause: an absentee queue was found by the message segment facilty to be in an inconsistent state, or a crawlout from the administrative ring occurred in the message segment facility.

Default action: prints a message and returns to command level.

Restrictions: since the message segment facility is used only for system services such as absentee queues, the user should not attempt to handle this condition.

Data structure: none.

mme1, mme2, mme3, mme4 (hardware)

Cause: the user attempted to execute the processor instruction mme*n*, where *n* is 1, 2, 3, or 4.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: the debug command uses the mme2 condition to implement breakpoints. Thus, the user encounters problems if he attempts to set breakpoints in a program that handles the mme2 condition.

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.

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.

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.

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.

Restrictions: none.

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.

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.

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.

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.

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.

Restrictions: none.

Data structure: none.

op_not_complete (hardware)

Cause: the processor failed to access memory within approximately 2 ms after its previous memory access.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: 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 a hardware error.

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.

Data structure: none.

overflow (hardware)

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: returning to the point where the signal occurred is not allowed since continued execution from this point is undefined.

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 condition mechanism.

Default action: prints a message and returns to command level.

Restrictions: none.

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.

Data structure: none.

program_interrupt

Cause: the user issued the program_interrupt (pi) command (described in the MPM Commands) 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 by the user when he issues a quit signal (presses the appropriate key on the terminal, e.g., ATTN, BRK, etc.).

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

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. In addition, a program with a quit handler is more difficult to debug since a bug in the quit handler might make it impossible to interrupt the execution of the program. 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.

Data structure: none.

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.

Data structure: the standard PL/I data structure.

record_quota_overflow (hardware)

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.

Data structure: none.

seg_fault_error (hardware)

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's access to the segment has been revoked.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

simfault_nnnnnn (hardware)

Cause: the user attempted to use a null pointer; i.e., a pointer with a segment number of -1 (2's complement) and an offset of nnnnnn. The offset is mapped into the six-character string <u>nnnnnn</u> that makes up part of the condition name.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: if a user references through a null pointer with no offset modification, the condition simfault_000001 is signalled.

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: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

Data structure: the standard PL/I data structure.

stack

Cause: the user attempted to make a reference within the last four pages of the stack.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: the any_other condition handlers should pass this on.

storage (PL/I)

Cause: 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 allocation is retried.

Restrictions: none.

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.

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: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

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.

Data structure: the standard PL/I data structure.

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: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

Data structure: the standard PL/I data structure.

timer_manager_err

Cause: the event channel on which the timer_manager_ subroutine would go to sleep could not be created; or the ipc_$block entry point returned a nonzero status code when the timer_manager_ subroutine went to sleep on it. Either internal static storage for the timer_manager_ subroutine has been destroyed or the system is about to crash. This condition is also signalled if the timer_manager_ subroutine is called in a ring other than that in which the process was created, indicating a programming error in the calling procedure.

Default action: prints a message and returns to command level.

Restrictions: the user should only attempt to handle this in a handler for otherwise unclaimed signals.

Data structure: none.

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.

Data structure: the standard PL/I data structure.

truncation (hardware)

Cause: the user executed an extended instruction set instruction to move string data with the truncation bit set and the target string was not large enough to contain the source string; or bit strings were being combined to the left or right (also EIS instructions) and there was not enough room to hold the combined string.

Default action: prints a message and returns to command level.

Restrictions: none.

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.

Data structure: the standard PL/I data structure.

underflow (hardware)

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.

Data structure: none.

Note: before the underflow condition is signalled 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.

Data structure:

```
declare invalid_label label;
```

where invalid_label is the invalid label to which this transfer was attempted.

zerodivide (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: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

Data structure: the standard PL/I data structure.

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.

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.

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. "Simulated Faults" below.)

Simulated Faults

By convention, the segment number -1 (in 2's complement form) is reserved for software simulated faults. The segment number is a dummy; i.e., no Multics segment ever has it. Any attempt to reference that segment number results in the out-of-bounds subcondition of the illegal procedure fault. When this fault occurs, the fault interceptor signals (in the ring where the fault occurred) the simfault_nnnnnn condition, where nnnnnn is the offset portion of the segment address that caused the fault. This convention provides an additional 256K faults, the first 128K of which are reserved for system use. The remaining 128K faults are available for user programs.

One of the software simulated faults reserved for system use is currently assigned. An offset of 1 (simfault_000001) is defined as the null pointer value for the PL/I pointer data type. Thus, the null pointer has the value (in 2's complement form) of -1¦1. An inadvertent reference by a user to a null pointer cannot produce an address with an offset of 1. In many cases, the null pointer is modified by an incremental offset. Thus, a null pointer modified by an offset of 22 (octal) produces the condition simfault_000023. Users who receive a message on their terminal indicating that a simfault occurred should check for inadvertent use of a null pointer.

Process Termination Fault

By convention, the segment number -2 (in 2's complement form) is reserved for the process termination fault. Any reference to that segment number causes the referencing process to be terminated. The offset portion of the segment address can be used to indicate the reason for the termination. Of the 256K possible offsets, the last 128K are reserved for interpretation by the system. The first 128K are available for user programs. Any offset currently recognized by the system is interpreted in a message printed on the user's terminal after the process is terminated.

SECTION VIII

BACKUP AND RETRIEVAL

The Multics backup system augments the reliability of the online storage system. It ensures 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. At the same time it produces a map indicating the segments and directories that have been dumped. The frequency of dumping and the length of time for which tapes are kept are determined at individual locations.

2. reloading

   Reloading is the recovering of segments and directories that have been dumped. Reloading of individual segments and directories can occur during normal Multics operation (retrieving). The entire contents of the online storage system can be reloaded after a system crash so that operation of the system can resume.

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. During each of the three modes, those portions of the hierarchy specified by a control segment are searched. The contents of this control segment are determined at each site. Usually, only two subdirectories of the root directory are not searched. One of these, >system_library_1, is always re-created 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.

By convention, the process named Backup.SysDaemon controls the incremental and consolidated dumps and the process named Dumper.SysDaemon controls complete dumps. The two processes can run simultaneously, without interfering with each other.

Incremental Dumps

Incremental dumping is the principal technique used to keep the backup system 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. This criterion is easily determined by comparing the date-time-modified and the date-time-dumped attributes found in the branch for any given segment or directory. Immediately after dumping, the incremental dumper resets the date-time-dumped attribute. 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. Furthermore, the consolidated dump also picks up segments and directories modified more recently than the last incremental dump. Hence, a consolidated dump performs the work of an incremental dump as well.

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. During a complete dump, the date-time-dumped attribute is not reset. Therefore, complete dumps do not interact in any way with incremental or consolidated dumps.

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.

RELOADING

The segment and directory recovery mechanism used by the backup system consists of a group of programs known as the reloader/retriever. The reloader/retriever is used to recover segments from tapes produced by any of the dumps. Retrieving occurs during normal Multics operation, while reloading occurs prior to normal Multics operation.

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 which dump tape contains a suitable copy of a given segment.

Once an estimate has been made as to which dump tape contains the desired copy, this estimate can be verified by examining the corresponding dump map. 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 reloaded. A directory for which the subtree is not reloaded contains only the links and access control information associated with the directory itself.

In special cases, a user can specify that the segment or 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. A directory subtree can be relocated at any point at the same level in the hierarchy (i.e., the number of greater-than characters in the pathname of the directory cannot change).

Normally, the most recent copy of an entry on the specified tape is retrieved. However, the user can specify that the first occurrence or an occurrence at some specified date is to be retrieved instead.

APPENDIX A

ASCII CHARACTER SET

PRINTING GRAPHIC CHARACTERS

The printing graphic characters are the uppercase alphabet, the lowercase alphabet, digits, and a set of special characters. The special characters are listed below.

!	exclamation point
"	double quote
#	number sign
$	dollar sign
%	percent
&	ampersand
´	acute accent
(	left parenthesis
)	right parenthesis
*	asterisk
+	plus
,	comma
-	minus
.	period
/	right slant
:	colon
;	semicolon
<	less than
=	equals
>	greater than
?	question mark
@	commercial at
[	left bracket
\	left slant
]	right bracket
^	circumflex
_	underline
`	grave accent
{	left brace
¦	vertical line
}	right brace
~	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.

The characters designated as unused are specifically reserved and can be assigned definitions at any time. Until defined, unused control characters are output 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.

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. This character is defined not to appear in a canonical string. See "Typing Conventions" for a description of canonical form.

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. This character is defined not to appear in a canonical string.

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 is defined not to appear in a canonical string.

RRS Red ribbon shift. ASCII SO (octal 016) is used for this character.

BRS 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 and cannot appear in a canonical character string. ASCII DEL (octal 177) is used for this character.

NONSTANDARD CONTROL CHARACTER

One control character, NUL, is recognized under certain conditions by all Device Interface 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 exactly as PAD. This character cannot appear in a canonical character string. 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	SUB	032
STX	002	DLE	020	NAK	025	ESC	033
ETX	003	DC1	021	SYN	026	FS	034
EOT	004	DC2	022	ETB	027	GS	035
ENQ	005	DC3	023	CAN	030	RS	036
				EM	031	US	037

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

APPENDIX B

TERMINAL CHARACTERISTICS

This appendix lists supported terminal types and escape conventions for various terminals that can be used to access the Multics system. For user convenience, terminals should support the full (128 characters) ASCII character set on input and output. Escape conventions have been provided for terminals that do not have a full ASCII character set. (See "Escape Conventions on Various Terminals" below.)

SUPPORTED TERMINAL TYPES

The terminals mentioned below are not the only terminals supported by Multics. They are, however, representative of the types of terminals that can be used to access the Multics system.

1. Device similar to an IBM Model 1050
2. Device similar to an IBM Model 2741
3. Device similar to a Teletype Model 37
4. Device similar to a GE TermiNet 300 or 1200
5. Device similar to an Adage, Inc. Advanced Remote Display Station (ARDS)
6. Device similar to an IBM Model 2741 with correspondence keyboard and 015 typeball
7. Device similar to a Teletype Models 33 or 35
8. Device similar to a Teletype Model 38
9. Device similar to a Computer Devices Inc. (CDI) Model 1030 or a Texas Instruments (TI) Model 725 or a device with an unrecognized answerback or a device without an answerback (ASCII device)

ESCAPE CONVENTIONS ON VARIOUS TERMINALS

The following paragraphs below list escape conventions for some of the terminals that can be used to access the Multics system. See also "Typing Conventions" in Section III, for information on keyboard input and output conventions. In general, the conventions described there apply to logging in and out as well as to all other typing.

IBM 1050 and IBM 2741

Each typeball used requires a different set of escape conventions.

With the EBCDIC typeball number 963, the following non-ASCII graphics are considered to be stylized versions of ASCII characters:

¢	(cent sign)	for	\	(left slant, software escape)
'	(apostrophe)	for	´	(acute accent)
¬	(negation)	for	^	(circumflex)

The following escape conventions have been chosen to represent the remainder of the ASCII graphics.

¢'	for	`	(grave accent)
¢<	for	[	(left bracket)
¢>	for	]	(right bracket)
¢(	for	{	(left brace)
¢)	for	}	(right brace)
¢t	for	~	(tilde)

With the correspondence typeball number 029, the following non-ASCII graphics are considered to be stylized versions of ASCII characters.

¢	(cent sign)	for	\	(left slant)
'	(apostrophe)	for	´	(acute accent)
±	(plus-minus)	for	^	(circumflex)

The following escape conventions have been chosen to represent the remainder of the ASCII graphics.

¢(	for	<	(less then)
¢)	for	>	(greater than)
¢l	for	[	(left bracket)
¢r	for	]	(right bracket)
¢:	for	!	(exclamation point)
¢t	for	~	(tilde)
¢'	for	`	(grave accent)
¢/	for	¦	(vertical bar)

NOTE: The left and right braces ({ and }) must be input using octal escapes (¢173 and ¢175) when using the correspondence typeball.

Teletype Models 33 and 35

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 otner 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 (^)

APPENDIX C

PUNCHED CARD INPUT AND OUTPUT

PUNCHED CARD INPUT

Each deck must begin with two (or more) keypunched control cards: an access_id card (that may extend to several cards if the information is too long to fit on one card) and a deck_id card. These cards are used to identify the submitter to the Multics system and describe the deck name and punch format. 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. No special access need be given to the user's directory for any system process. 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 MPM Commands.)

Card image segments must be copied from the System Pool Storage within a reasonable time, as these segments are periodically deleted.

Control Card Formats

The access_id card has the following format:

```
PERSON_ID.PROJECT_ID  ACCESS_CLASS;
```

where:

1. PERSON_ID — is the registered name of the submitter. Only this person can read the card image segment from the pool. A PERSON_ID of "*" is not allowed.

2. PROJECT_ID — is the project name of the submitter. The Person_id is separated from the Project_id by a period. Specifying the Project_id is optional.

3. ACCESS_CLASS — is the access class of this data. The ACCESS_CLASS field may contain embedded blanks and commas. If the ACCESS_CLASS is too long to fit on one card, it may be split between an access class component and extended onto another card. When this happens, the last nonblank character of all but the final access_id card must be a comma (,). The final access_id card must have a semicolon as the last nonblank character.

Omission of the ACCESS_CLASS field indicates the system_low access class at sites where system_low is unnamed. Failure to correctly format or specify the access_id information may result in the inability of the user to access the data from the System Pool Storage.

The deck_id card has the following format:

```
DECK_NAME  PUNCH_FORMAT
```

where:

1. DECK_NAME is the name used 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.

2. PUNCH_FORMAT is the punch code conversion to use in reading the card deck. It must be MCC, VIIPUNCH, or RAW. If this field is omitted, MCC format is assumed.

If name duplications are encountered then there may be more than one deck in System Pool Storage whose first component is DECK_NAME. The copy_cards command retrieves all of these copies when invoked.

The control cards should be produced on a standard IBM 029 key punch. The fields on the control cards are free format with spaces separating the fields. The only restrictions are:

1. the first access_id card must contain the PERSON_ID, plus the PROJECT_ID and first component of the ACCESS_CLASS if the PROJECT_ID and ACCESS_CLASS are specified.

2. the deck_id card must contain the DECK_NAME, plus the PUNCH_FORMAT if the PUNCH_FORMAT is specified.

All characters on the control cards are mapped to lower case except those immediately following an escape character (backslash or cent sign). For example \MY_\DECK.PL1 is mapped to My_Deck.pl1.

Example

Suppose user Doe working on project Proj, wishes to read a FORTRAN source deck into a segment called alpha.fortran, with an access class of "proprietary, my_company". The access_id card is:

```
\DOE.\PROJ   PROPRIETARY, MY_COMPANY;
```

and the deck_id card is:

```
ALPHA.FORTRAN   MCC
```

where MCC is the format of the data cards. The control cards followed by the data cards are submitted to operations personnel for reading. When the cards have been read into Multics by operations personnel, the submitter should log in as Doe.Proj with an access authorization of "proprietary, my_company" and issue from the terminal the command:

```
copy_cards  alpha.fortran
```

to copy the deck into the working directory. If the copy does not succeed, then an error message explains the problem. The user may need to check with operations to correct the problem.

Deck Size

Decks must not exceed the maximum length of a Multics segment. It is wise to limit decks to single boxes of cards, although more precise counts can be made. For raw reading, the actual maximum is 9,792 cards. For Multics card codes, the actual maximum depends on the number of characters actually read since trailing blanks on cards are ignored. Assuming all 80 columns are punched on each card, the maximum would be 13,055 cards. For 7punched decks, the length of the created segment depends on the length of the original data. The typical 7punch card represents 22 words, but it may represent as many as 4,096 words if the original data contained that many consecutive words of identical contents.

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 deck produced as a result of the dpunch command has some additional punched cards before and after the requested data. These cards are used to identify the deck and its owner. They 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
SEPARATOR CARD		
User's Data	-	punched in the requested format
END OF DECK	-	punched in flip card format
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 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 will 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 correctly.

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. cccccccccccc — 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	
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	

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
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 accute accent into EBCDIC apostrophe
050	(	(	4D	12-8-5	
051	)	)	5D	11-8-5	
052	*	*	5C	11-8-4	
053	+	+	4E	12-8-6	
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	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

ASCII Code Value	ASCII Meaning	Corre-sponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
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	
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	

ASCII code values are in octal; EBCDIC code values are in hexadecimal

ASCII Code Value	ASCII Meaning	Corre- sponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
127	W	W	E6	0-6	
130	X	X	E7	0-7	
131	Y	Y	E8	0-8	
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	
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	

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
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 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

APPENDIX D

STANDARD DATA TYPE FORMATS

This appendix describes the representation of Multics standard data types. See "Subroutine Calling Sequences" in Section II of the MPM Subsystem Writers' Guide for a discussion of data descriptors. 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.

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.

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 of the form:

```
                  3
 0   7 8          5
|  e  |     m     |
```

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $1/2\leq|m|<1$.

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.

```
 0    7 8     p+8
_|____|_|_______|_
 |  e   |   m   |
```

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $1/2\leq|m|<1$.

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 that has an even address.

```
                                  7
 0    7 8                         1
 |  e   |             m           |
```

The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $1/2\leq|m|<1$.

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.

```
 0    7 8                       p+8
 |  e   |             m           |
```

The value 0 is represented as m=0 and e=-128. For all other values, m satisfies $1/2\leq|m|<1$.

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 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 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 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 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 (descriptor type 9)

A real, fixed-point, decimal 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, decimal datum is aligned on a word boundary and occupies an integral number of words, the last of which can contain unused bytes.

```
 _____________________________
|s | d1 | d2 | ... | dp |
```

Real Floating-Point Decimal (descriptor type 10)

A real, floating-point, decimal 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 characters of the form:

$$\underbrace{\boxed{s \mid d1 \mid d2 \mid \;\ldots\; \mid dp}}_{m}\ \underbrace{\boxed{0 \mid \quad}}_{e}$$

The exponent e is right justified within the last 9-bit character and the unused bit is zero. The value 0 is represented by m=o and e=+127.

An unpacked, decimal datum is aligned on a word boundary and occupies an integral number of words, some bytes of which can be unused.

Complex Fixed-Point Decimal (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 represents the real part of the complex value, and the second represents the imaginary part of the complex value.

An unpacked, complex, decimal datum is aligned on a word boundary and occupies an integral number of bytes, some of which can be unused.

Complex Floating-Point Decimal (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 represents the real part of the complex value and the last represents the imaginary part of the complex value.

An unpacked, complex, decimal datum is aligned on a word boundary and occupies an integral number of bytes, some of which can be unused.

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, stored in a pair of 36-bit words the first of which has an even address.

```
         1 1   2 2   2 3   3 3  5 5   5 5  6 6   6 6   7
 0  2 3  7 8   0 1   9 0   5 6  3 4   6 7  2 3   5 6   1
|    | s  |  r  |     | 43  | w  |  0  | b  |     |  0  |
```

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.

```
           1 1          3
 0   5 6   7 8          5
|  b  |  s  |      w     |
```

Offset (descriptor type 14)

An offset datum (always unpacked) is represented by a word offset w, and a bit offset b, stored in a single 36-bit word.

```
               1 1   2 2   2     3
 0             7 8   0 1   6     5
|      w       | 0   |    |  b   |
```

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 block immediately 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 block immediately 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 decimal arithmetic or non varying string datum is aligned on a 9-bit byte boundary.

An unpacked structure occupies an integral number of words.

Area (descriptor type 18)

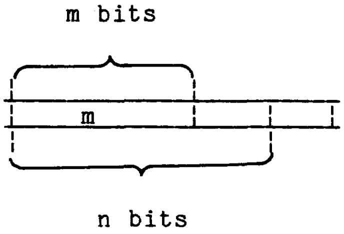
An area datum (always unpacked) whose declared size is k occupies k words of storage, the first of which has an even address. The maximum space available for allocations within the area occupies k minus 24 words. The number of words required for cards allocations is 2+ (2**m) where m is the event power of 2 that exceeds the size of the items being allocated.

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, aligned integer followed by a nonvarying bit string of length n.



The length of the current value is m. 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.

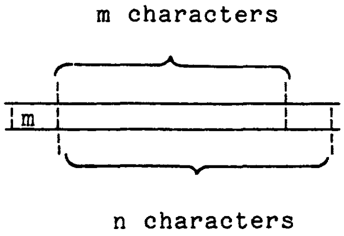
Character String (descriptor type 21)

A character string (packed or unpacked) whose length is n occupies n consecutive 9-bit bytes. Each byte contains a single 7-bit ASCII character right justified within the byte. The two unused bits must be zero.

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, unaligned integer followed by a nonvarying character string of length n.



The length of the current value is m.

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.

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).

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
10	real floating-point decimal
11	complex fixed-point decimal
12	complex floating-point decimal
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

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 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:

exec_com_stream_N — is the switch attached by the exec_com command using the attach command line where N is a unique sequence number assigned by the exec_com command. The switch user_input is attached to this switch through the syn_ I/O module.

file_output_stream_ — is the switch attached by the file_output command. The switch user_output is attached to this switch through the syn_ I/O module.

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.

RESERVED SEGMENT NAMES

By convention, the following segment names are reserved. Those 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 (described in the MPM Commands).

Person_id.con_msgs is the segment used by the message facility (see the send_message command in the MPM Commands).

Person_id.mbx is the segment used by the mail command (described in the MPM Commands).

Person_id.memo is the segment used by the memo command (described in the MPM Commands).

Person_id.motd is the segment used by the print_motd command (described in the MPM Commands).

Person_id.profile is the segment used by the abbrev command (described in the MPM Commands).

Those maintained in the process directory are:

combined_linkage_N.jk is the user's linkage segment for ring number N (1<=N<=7). jk is a two digit sequence number. This segment also contains internal static storage.

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 MPM Subroutines).

stack_N is the user's automatic storage area for ring number N (1<=N<=7).

system_free_N_ is the free storage area used by system commands for ring number N (1<=N<=7).

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 IV.)

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	

The listing segment suffix is:

list   is the suffix on printed output listing segments produced by compilers, the assembler, and the binder.

Other special suffixes are:

absin   is the input segment suffix for an absentee process.

absout   is the default output segment suffix for an absentee process.

apl   is the suffix on the segment containing a saved workspace from the apl command.

archive   is the suffix on the segment created by the archive command.

bind   is the suffix on the input control segment for the binder.

ec   is the suffix on the input segment to the exec_com command.

gcos   is the suffix on a segment that is in GCOS standard system format.

info   is the suffix on a segment, in >documentation>info_segments, for use with the help command.

mbx   is the suffix on any mailbox segment that the user wants to create.

ms   is the suffix on an administrative ring message segment.

qedx   is the suffix concatenated by the qedx command to the entryname of a segment containing qedx instructions.

runoff   is the input segment suffix to the runoff command.

runout   is the output segment suffix from the runoff command.

RESERVED OBJECT SEGMENT ENTRY POINT

By convention, the following entry point definition in object segments is reserved.

symbol_table — is the entry point definition which provides the address of the symbol table produced by the pl1 or fortran commands.

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

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 MPM Subroutines). Any magnetic tape not written in the standard format described here is not a Multics standard tape.

STANDARD TAPE FORMAT

The first record on the tape following the beginning of tape (BOT) mark is the tape label record. Following the tape record is an end of file (EOF) mark. Subsequent reels of a multireel sequence also have a tape label followed by EOF. (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 record consists of a 1024-word (36864-bit) data space enclosed by an 8-word header and an 8-word trailer. The total record length is then 1040 words (37440 bits). The header and trailer are each 288 bits. This physical record requires 4680 frames on 9-track tape and 6240 frames on 7-track tape. This is approximately 5.85 inches on 9-track tape at 800 bpi and 7.8 inches on 7-track tape at 800 bpi, not including interrecord gaps. (Record gaps on 9-track tapes are approximately 0.6 inches and on 7-track tapes are approximately 0.75 inches, at 800 bpi.)

For 1600 bpi 9-track tape, the record length is approximately 2.925 inches (with an interrecord gap of approximately 0.5 inches).

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 file, 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. (This should be a constant equal to 36864.)

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).
20-26	Reserved.
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 Appendix G, "Standard Checksum," 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 two types of administrative records: a tape label record; or, an EOR record.

The administrative records are of standard length: 8-word header, 1024-word data area, and 8-word trailer.

The tape label record is written in the standard record format. The data space of the tape label record contains:

Words 0-7: 32-character ASCII installation code. This identifies the installation that labelled the tape.

Words 8-15: 32-character ASCII reel identification. This is the reel identification by which the operator stores and retrieves the tape.

remaining: a padding pattern.

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 1).

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) and 1600 bpi (recorded in PE mode).

DATA PADDING

The padding bit pattern is used to fill administrative records and the last data record of a reel sequence.

WRITE ERROR RECOVERY

Multics standard tape error recovery procedures differ from the past standard technique in that no attempt is made to backspace the tape on write errors. If a data alert occurs while writing a record, the record is rewritten. If an error occurs while rewriting the record, that record is again rewritten. Up to 64 attempts can be made to write the record. No backspace record operation is performed.

The above write error recovery procedure is applied to both administrative records and data records.

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.

APPENDIX G

STANDARD CHECKSUM

The checksum described in this appendix is the standard Multics technique for computing a full word checksum on the Honeywell 6180 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:

```
       ldi     =o004000,dl    inhibit overflow fault
       sti     indics         save indicators
       lda     0,dl           initialize "a" to zero
       eax1    0              count locations in x1

loop:  ldi     indics         restore indicators
       awca    word,1         add with carry to checksum
       sti     indics         save indicators (they get
                              clobbered by cmpx1)
       alr     1              rotate "a" left
       eax1    1,1            count 1 location and
       cmpx1   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
```

INDEX

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS PROGRAMMERS' MANUAL – REFERENCE GUIDE

ORDER NO. AG91, Rev. 1

DATED DECEMBER 1975

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. □

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

Honeywell

The Other Computer Company:

Honeywell

HONEYWELL INFORMATION SYSTEMS

14841
1.5176
Printed in U.S.A.

In the U.S.A.: 200 Smith Street, MS 061, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario