HONEYWELL

DPS 6 & LEVEL 6 GCOS 6 MOD 400 SYSTEM CONCEPTS

SOFTWARE

LEVEL 6 & DPS 6
GCOS 6 MOD 400
SYSTEM CONCEPTS

SUBJECT

System concepts for GCOS 6 MOD 400

SPECIAL INSTRUCTIONS

This manual supersedes the *GCOS 6 MOD 400 System Concepts* manual, Order No. CB20-01, dated February 1980, and its addendum CB20-01A, dated October 1980.

SOFTWARE SUPPORTED

See the *MOD 400 Guide to Software Documentation* for information about Executive releases supported by this manual.

ORDER NUMBER

CZ03-00

December 1982

Honeywell

PREFACE

This manual is written for all users of the MOD 400 system.

It will prove particularly informative to those responsible for building MOD 400 systems and those who design application programs and/or system functionality other than that supplied by Honeywell.

This manual contains a general description of the way in which processing is performed on MOD 400 systems. It presents a discussion of the MOD 400 Executive in terms of its design concepts and processing functionality. Not discussed are such topics as equipment lists; available software, and supporting manuals. No detailed procedural information is discussed; several procedures are, however, outlined.

The major topics discussed are:

- File system, including file and pathname concepts, file protection, and buffering operations.

- System access path including login, user registration, and the command environment.

- Execution environment, including a description of tasks, task groups, memory usage, and bound units.

- Task execution, including priority levels, logical resource numbers, and intra/inter task communication.

- Deferred processing facilities, including deferred group processing, deferred print/punch processing, and the queuing and transacription of report files.

Honeywell disclaims the implied warranties of merchantability and fitness for a particular purpose and makes no express warranties except as may be stated in its written agreement with and for its customer.

In no event is Honeywell liable to anyone for any indirect, special or consequential damages. The information and specifications in this document are subject to change without notice.

- Backup and recovery facilities, including the backup and restoration of disk files, the preservation of the execution environment during a power failure, the recovery of files at the record level, and the recovery and restart of task groups.

Although no manual is prerequisite to this manual, you may find it convient to have read the Software and Documentation Directory.

The following symbols are used in this manual:

- Square brackets [ ] indicate an optional entry.
- Braces { } enclose information from which the user must make a choice.
- Lowercase letters (e.g., id) indicate a symbolic variable whose exact value must be supplied by the user.
- Uppercase letters (e.g., MEMPOOL) indicate commands or directives that must be reproduced exactly as shown.
- The character (delta) or the word "blank" indicates that the entry so identified should be blank.

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

Level	Heading Format
1 (highest)	ALL CAPITAL LETTERS, UNDERLINED
2	Initial Capital Letters, underlined
3	ALL CAPITAL LETTERS, NOT UNDERLINED
4	Initial Capital Letters, not underlined

MANUAL DIRECTORY

The following publications constitute the GCOS 6 MOD 400 manual set. Refer to the "Software/Manual Directory" of the Guide to Software Documentation for the current revision number and addenda (if any) of relevant release-specific publications.

Manuals are obtained by submitting a Honeywell Publications Order Form to the following address:

Honeywell Information Systems Inc.
47 Harvard Street
Westwood, MA 02090

Attn: Publications Services

Honeywell software reference manuals are periodically updated to support enhancements and improvements to the software. Before ordering any manuals, you should refer to the Guide to Software Documentation to obtain information concerning the specific edition of the manual that supports the software currently in use at your installation. If you use the four-character base publication number to order a document, you will receive the latest edition of the manual. The Publications Distribution Center can provide specific editions of a publication only when supplied with the seven- or eight-character order number listed in the Guide to Software Documentation.

Honeywell applications software packages, such as INFO 6, TOTAL 6, and TPS 6, provide specialized services. Contact your Honeywell representative for information concerning the availability of applications software and supporting documentation.

Base Publication Number	Manual Title
CZ01	GCOS 6 MOD 400 Guide to Software Documentation
CZ02	GCOS 6 MOD 400 System Building and Administration
CZ03	GCOS 6 MOD 400 System Concepts
CZ04	GCOS 6 MOD 400 System User's Guide
CZ05	GCOS 6 MOD 400 System Programmer's Guide - Volume I
CZ06	GCOS 6 MOD 400 System Programmer's Guide - Volume II
CZ07	GCOS 6 MOD 400 Programmer's Pocket Guide
CZ09	GCOS 6 MOD 400 System Maintenance Facility Administrator's Guide
CZ10	GCOS 6 MOD 400 Menu Management/Maintenance Guide
CZ15	GCOS 6 MOD 400 Application Developer's Guide
CZ16	GCOS 6 MOD 400 System Messages
CZ17	GCOS 6 MOD 400 Commands
CZ18	GCOS 6 Sort/Merge
CZ19	GCOS 6 Data File Organizations and Formats
CZ20	GCOS 6 MOD 400 Transaction Control Language Facility
CZ21	GCOS 6 MOD 400 Display Formatting and Control
CZ34	GCOS 6 Advanced COBOL Reference
CZ35	GCOS 6 Advanced COBOL Quick Reference Guide
CZ36	GCOS 6 BASIC Reference
CZ37	GCOS 6 BASIC Quick Reference Guide
CZ38	GCOS 6 Assembly Language (MAP) Reference
CZ39	GCOS 6 Advanced FORTRAN Reference
CZ40	GCOS 6 Pascal User's Guide
CZ41	GCOS 6 RPG-II Reference
CZ47	Data Entry Facility-II User's Guide
CZ48	Data Entry Facility-II Operator's Quick Reference Guide
CZ52	DM6 I-D-S/II Programmer's Guide
CZ53	DM6 I-D-S/II Data Base Administrator's Guide
CZ54	DM6 I-D-S/II Reference Card
CZ59	Level 6 to Level 6 File Transmission Facility User's Guide
CZ60	Level 6 to Level 66 File Transmission Facility User's Guide
CZ61	Level 6 to Level 62 File Transmission Facility User's Guide
CZ62	BSC Transport Facility User's Guide
CZ63	2780/3780 Workstation Facility User's Guide
CZ64	HASP Workstation Facility User's Guide

Base Publication Number	Manual Title
CZ65	Programmable Facility/3271 User's Guide
CZ66	Remote Batch Facility/66 User's Guide
CZ71	DM6 TP Development Reference
CZ72	DM6 TP Application User's Guide
CZ73	DM6 TP Forms Processing

In addition, the following publications provide supplementary information:

AS22	Level 6 Models 6/34, 6/36, and 6/43 Minicomputer Handbook
AT97	Level 6 Communications Handbook
CC71	Level 6 Minicomputer Systems Handbook
CD18	Level 6 MOD 400/600 Online Test and Verification Operator's Guide
FQ41	Writable Control Store User's Guide

Users should be aware that a Software Release Bulletin accompanies each software product ordered from Honeywell. You should consult the Software Release Bulletin before using the software. Contact your Honeywell representative if a copy of the Software Release Bulletin is not available.

CONTENTS

CONTENTS

CONTENTS

Page

ILLUSTRATIONS

TABLES

SYSTEM
CHARACTERISTICS

Section 1
SYSTEM CHARACTERISTICS

GCOS 6 MOD 400 software is a disk-based operating system that supports multitasking, real-time, or data communications applications in one or more online streams. In addition, program development or other batch-type applications can be performed concurrently in a single batch stream.

GCOS is a multifunctional system capable of supporting a variety of processing functions. You can develop and execute applications software, perform forms data entry, transmit files to other DPS 6/Level 6 computers, and enter jobs for execution at remote sites.

The system can be configured to process different functional applications concurrently. For example, you can run your own applications, utilize other system functionality such as the data collection capability, and communicate with a host processor at the same time.

OPERATING FACILITIES

MOD 400 supports multiprogramming, the concurrent execution of multiple tasks running under one or more task groups. Each task group owns the resources necessary for execution of an application program (one or more related tasks). The task group runs independently in its own operating environment while it shares the resources of the system.

If you define the environment to run more than one application task group concurrently, you are utilizing multiprogramming. In this environment you can execute each task in a task group sequentially or concurrently, which is multitasking. You can run multiple online task groups concurrently with a single batch task group.

The number of task groups that can run is limited only by the amount of memory available. Concurrently executing task groups can occupy independent dedicated memory areas, or they can contend for space within a memory pool. When one task group is deleted, the released memory is available to other task groups in the same pool. MOD 400 allocates memory dynamically from pools and can relocate programs at load time. Once a task group requests execution, its tasks are dispatched according to their assigned priority levels. When more than one task shares a priority level, tasks are serviced in a round-robin fashion.

Use of disk files by multiple independent users is facilitated by the arrangement of File System entries (directories and files) in a tree-structured hierarchy. Each directory or file is identified by a pathname that indicates the path from the root directory of the hierarchical structure of the particular directory or file. File reference can be simplified through the use of pathnames relative to a working directory that indicates a user's current position in the File System hierarchy. Access to sharable files and devices is controlled by file attributes and concurrency procedures.

SOFTWARE FACILITIES

MOD 400 offers you a comprehensive set of software components that perform a wide variety of functions. The following paragraphs briefly describe these software components.

System Control Software

System control software includes:

- Task Manager: Handles the disposition of tasks within the system and responds to requests placed against tasks. The Task Manager processes requests to activate tasks; returns control to interrupted tasks; and synchronizes, suspends, and terminates tasks.

- Clock Manager: Handles all requests to control tasks based on real-time considerations and responds to requests for the time of day and date in ASCII format.

- Segment Manager/Swapper: Controls the allocation of swap pool memory and swap file space. Swaps tasks out when swap pool memory is required and swaps them back when the memory is available.

- Memory Manager: Controls dynamic requests for memory or the return of memory to group work segments. Also controls the allocation of all memory in non-swapped pools and of tasks groups assigned to the swap pool.

- Trap Manager: Handles the transfer of execution control from an executing program to a predefined trap location when a trap (a special condition such as a hardware error) occurs. The Trap Manager handles system traps and allows a task group to connect its own trap routines for specific traps.

- Operator Interface Manager: Manages all messages sent simultaneously by multiple task groups to the operator terminal or from the operator terminal to a task group.

- Loader: Loads the root and overlays of a bound unit dynamically from a disk.

- Listener: Monitors a selected set of local and remote terminals. If you enter a log-in command requesting access to the system at one of the terminals, the Listener causes a task to be spawned for you.

- Command Processor: Processes all commands. The Command Processor is the lead task of the batch task group and can be the lead task of an online task group.

- Message Facility: Provides a means for intertask and intratask group communications. The Message Facility uses mailboxes as structures for sending and receiving messages.

- Menu Subsystem: Provides an alternative means to the Command Processor for communicating with the Executive.

File System Software

MOD 400 provides software to handle Input/Output (I/O) functions of each of the supported devices. The File System software is designed to work in conjunction with the data management conventions established for each device. The File System software is available through system commands or, for an Assembly language program, through system service macro calls.

Utility Software

The system provides a comprehensive set of utility programs for performing frequently used programming functions. The system programs used by MOD 400 for the various utility functions are invoked by system commands.

Program Development Software

MOD 400 supports a large set of program preparation components, utilities, and debugging aids for applications development. Programming languages include Assembly language, PASCAL, FORTRAN, COBOL, and BASIC. A display formatting and control facility provides an effective method for developing, displaying, maintaining, and utilizing terminal display forms.

Data Communications Software

MOD 400 supports four levels of communications interface; terminals and/or remote host computers can be accessed through:

1. Sequential file interface of the File System software
2. Display formatting and control software
3. Physical I/O interface of the system
4. Various distributed systems facilities.

Specialized software components called Line Protocol Handlers (LPHs) support the different device classes and the various conventions established for data transfer.

Distributed Systems Software

MOD 400 supports various software packages that permit use of DPS 6/Level 6 in a distributed processing environment. Using the packages provided by the vendor, you can configure a DPS 6/ Level 6 as a host processor with specialized processing assigned to remote terminals. Alternatively, you can develop links with a remote host processor and distribute the total processing load between the DPS 6/Level 6 and the host processor.

FILE CONCEPTS

Section 2
FILE CONCEPTS

A file is a logical unit of data composed of a collection of records. The principal external devices available for storing files are:

- Disk devices (diskettes, cartridge disks, cartridge module disks, and mass storage units)
- Magnetic tape units.

These external devices are referred to as volumes (e.g., diskette volume, tape volume).

Various conventions have been established to identify and locate files stored on disk and magnetic tape. The conventions facilitate the orderly and efficient use of the stored data.

Unit record devices (such as card readers, card punches, paper tape reader/punches, and printers) also use the file concepts. However, since unit record devices cannot be used to store files, there is less need to establish conventions for identification and location. A unit record file is simply the data that is read or written at any one time.

DISK FILE CONVENTIONS

You must be able to specify an access path to any given file on a disk volume that contains multiple files. Files must therefore be organized on the volume in some predictable fashion. MOD 400 provides a set of volume organization conventions by which the system can locate any element that resides on the volume.

The principal elements of this organization, aside from the files themselves, are directories. The access path to any given element on a volume is known as a pathname.

Directories

Files on disk devices reside within a tree-structured hierarchy. The basic elements of this hierarchy are special files known as directories. The directories are used to point to the location of data files, which are the endpoints of the tree structure.

A directory on a disk volume is an index that contains the names and starting locations (sectors on the volume) of files or other directories (or both). The elements in the directory are said to be "contained in" or "subordinate to" the directory. Therefore, the organization of a disk volume is a multilevel structure. The complexity of the access path to any given element in the structure depends on the number of directories between the root and the desired element.

A directory structure is illustrated in Figure 2-1. The base directory on a volume is termed a root directory. In Figure 2-1 the root directory is VOL01. The root directory VOL01 points to two subordinate directories DIR1 and DIR2. The directories DIR1 and DIR2, in turn, point to the data files (FILEA, FILEB, FILEC, and FILED).

The root directory and other special types of directories are described in the following paragraphs.

ROOT DIRECTORY

There is a tree structure for each disk mounted at any given time. At the base of each tree structure is a directory known as the root directory. This is the directory that ultimately contains every element that resides on the volume either immediately or indirectly subordinate to it. The root directory name is the same as the volume identifier of the volume on which it resides. The directory VOL01 in Figure 2-1 is a root directory.

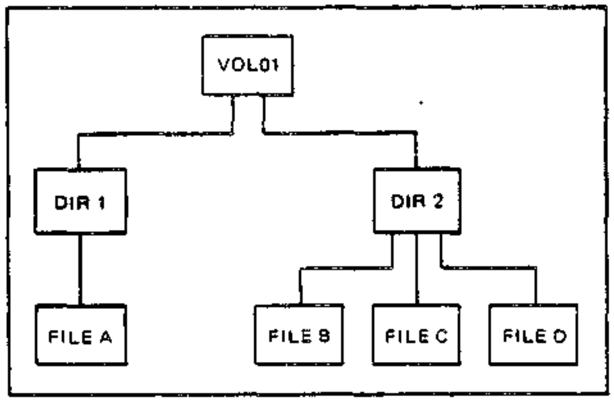


Figure 2-1. Example of Disk File Directory Structure

SYSTEM ROOT DIRECTORY

One or more disk root directories can be known to the system at any time during its operation. One of these, the System Root Directory (SRD), is required at all times. Files in the SRD all have pathnames starting with two greater-than signs (>>). The volume used by the operator to initialize the system establishes the SRD; the boot volume must contain the SRD. This volume also normally contains system programs, commands, and other routinely used elements. It must contain a number of directories and files that the system needs to perform its functions, including Z3EXECUTIVEL, SID, AID, HIS, and USER_REG. For more information, see the GCOS 6 System Building and Administration manual (Order No. CZ02).

USER ROOT DIRECTORY

The File System can recognize one User Root Directory (URD). Files in the URD have pathnames that start with a single greater-than character. The URD contains such items as UDD, LDD, MDD, FORMS, PROGS, and TRANS. For more information, see the System Building and Administration manual. The SRD and the URD can reside on different volumes or on the same volume. The installation can also support several user volumes that were created and used for the installation's own particular needs. They may contain user application programs and their associated data files, application program source and object code files, listing files, or anything else that a user might want to store, either temporarily or permanently.

INTERMEDIATE DIRECTORIES

When you first create a volume, it contains only a root directory. Within this directory you can create any additional directories required to satisfy the needs of your installation. Consider, for example, a volume that is to contain data used by two application projects, each of which has several people associated with it. Each person has one or more files of interest to him/her. The volume has been initialized and contains a root directory name. Two directories can be created subordinate to the root directory, each identified by the project name. Then, subordinate to these directories, a directory can be created for each person associated with each project.

The data files are all contained within the personal directories. This sample directory structure is illustrated in Figure 2-2.

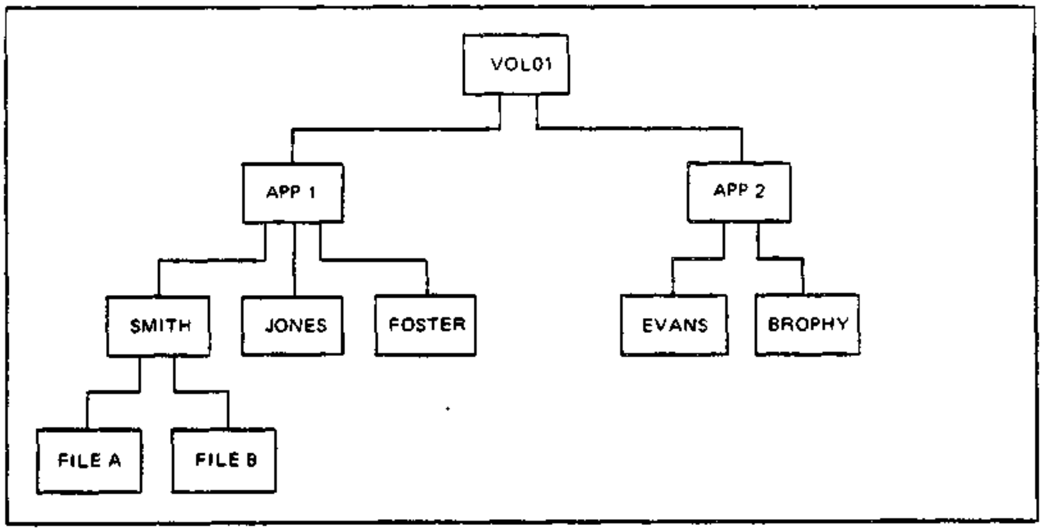


Figure 2-2. Sample Directory Structure

When the need for a user-created directory no longer exists, the directory can be deleted from the File System. The space it occupied, as well as the space occupied by its attributes in the immediately superior directory, is then available for reuse. A directory must be empty before it can be deleted; all directories and files subordinate to the one to be deleted must have been previously deleted by explicit commands.

WORKING DIRECTORY

The File System always starts at a root directory when it performs an operation on a disk file or a directory. At times the search for an element residing on a disk volume may traverse a number of intermediate directory levels before locating the desired element, and the File System must be supplied with the names of all of the branch points it must pass on the way. All the files of interest to a user doing work on the system are frequently contained in a single directory. This directory can be three or four or more levels deep in the structure. It is convenient to be able to refer to files in relation to a directory at some arbitrary level in the hierarchy rather than in relation to the root directory. The File System allows this to be done by recognizing a special kind of directory known as a working directory.

A working directory establishes a reference point that enables you to specify the name of a file or another directory in terms of its position relative to the working directory. If the access path of the working directory is made known to the File System, and if the desired element is contained in that directory, then the element can be specified by just its name. The File System then concatenates this name with the names of the elements of the working directory's access path to form the complete access path to the element.

LOCATIONS OF DISK DIRECTORIES AND FILES

The File System has total control over the physical location of space allocated to directories and files; you need never be concerned about where a directory or file resides on a volume. When a volume is first initialized, space is allocated to elements in essentially the order in which they are created. But, after the volume has been in use for some time, elements may have been deleted and the space they occupied made reusable. Then, when a new element is created, it is allocated the first available space. If more space is needed, it is obtained from the next free area.

NAMING CONVENTIONS

Each disk file and directory name in the File System can consist of the following American Standard Code for Information Interchange (ASCII) characters:

- Uppercase alphabetics (A through Z)
- Digits (0 through 9)
- Underscore (_)
- Hyphen (-)
- Period (.).

If lowercase alphabetic characters are used, they are converted to their uppercase counterparts.

The first character of any name must be alphabetic. The underscore character can be used to join two or more words that are to be interpreted as a single name (e.g., DATE_TIME). The period character followed by one or more alphabetic or numeric characters is normally interpreted as a suffix to a file name. This convention is followed, for example, by a compiler when it generates a file that is to be subsequently listed; the compiler identifies this file by creating a name of the form "FILE.L".

The name of a root directory or a volume identifier can consist of from one to six characters. The names of other directories and files can contain from 1 to 12 characters. The length of a file name must be such that any system-supplied suffix does not result in a name of more than 12 characters.

UNIQUENESS OF NAMES

Within the system at any given time, the access path to every element must be unique. This leads to the following rules:

1. Only one volume with a given volume_id can be mounted at any given time. (The system notifies you of an attempt to mount a volume having the same name as one already mounted.)

2. Within a given directory, every immediately subordinate directory name must be unique. (The Create Directory command notifies you of an attempt to add a duplicate directory name.)

3. Within a given directory, every file name must be unique. (The Create File command notifies you of an attempt to add a duplicate file name.)

Pathnames

The access path to any File System entity (directory or file) begins with a root directory name and proceeds through zero or more subdirectory levels to the desired entity. The series of directory names (and a file name if a file is the target entity) is known as the entity's pathname. The total length of any pathname, including all symbols, cannot exceed 57 characters. A working directory pathname, however, cannot exceed 44 characters.

SYMBOLS USED IN PATHNAMES

The following symbols are used to construct pathnames:

- Circumflex (^): Used exclusively to identify the name of a disk volume root directory. The circumflex is used in two forms. In one form it directly precedes the root directory name (e.g., ^VOL011). In the other it directly precedes a greater-than symbol (>) to refer to the root directory of the current working directory (e.g., ^>DIR1>FILEA becomes ^VOL011>DIR1>FILEA).

- Greater Than (>): Indicates movement in the hierarchy away from the root directory. The symbol is used to connect two directory names or a directory name and a file name. It can also be the first character of a pathname, in which case the element whose name follows the > symbol is immediately subordinate to the root directory of the user root volume (residing under the URD). Each occurrence of the > symbol denotes a change of one hierarchical level; the name to the right of the symbol is immediately subordinate to the name on the left. Reading a pathname from left to right thus indicates movement through the tree structure in a direction away from the root directory. If the root directory ^VOL011 contains a directory name DIR1, then the pathname of DIR1 is:

```
^VOL011>DIR1
```

If the directory named DIR1 in turn contains a file named FILEA, then the pathname of FILEA is:

```
^VOL011>DIR1>FILEA
```

- Two Consecutive Greater-Than Signs (>>): Specifies entities that are subordinate to the SRD.

  Honeywell-supplied programs make the following assumptions about directory assignments to the SRD and URD:

SRD	URD
Z3EXECUTIVEL	UDD
SID	LDD
AID	FORMS
HIS	MDD
USER_REG	PROGS
	TRANS

  SYSLIB1 and SYSLIB2 can reside in either directory.

  The correct way to refer to the directory SID, for example, is >>SID; the correct way to refer to UDD is >UDD.

- Less Than (<): Used at the beginning of a pathname to indicate movement from the working directory in a direction toward the root directory. Consecutive symbols can be used to indicate changes of more than one level; each occurrence represents one level change. When followed by elements of a relative pathname, those elements represent changes of direction away from the root directory. One or more of these symbols may precede only a relative pathname.

- ASCII Space Character: Used to indicate the end of a pathname. When represented in memory, a pathname must end with a space character.

The last (or only) element in a pathname is the name of the entity upon which action is to be taken. This element can be a device name, directory name, or a file name, depending on the function to be performed. For example, in the Create Directory command, a pathname specifies the name of a directory to be created. The last element of this pathname is interpreted by the command as a directory name; any names preceding the final name are names of superior directories leading to it. An analogous situation occurs in the Create File command, except that in this case the final pathname element is the name of a file to be created.

ABSOLUTE AND RELATIVE PATHNAMES

A full pathname contains all necessary elements to describe a unique access path to a File System entity, regardless of the type and location of the device on which it resides. The File System uses this form in referring to a directory or file. However, it is frequently unnecessary for you to specify all of these elements; the File System can supply some of them when the missing elements are known to it and the abbreviated pathnames are used in the appropriate context. An understanding of these conditions and contexts requires an understanding of absolute and relative pathnames. These subjects are described in the following paragraphs.

Absolute Pathname

An absolute pathname is one that begins with a circumflex (^) or one or more greater-than symbols (>). A pathname that begins with a circumflex is a full pathname. This form is used to locate directories and files that reside on a device other than that on which the system (the volume from which the system was initialized) is mounted. When an absolute pathname begins with one greater-than symbol, the first element named in the pathname is assumed to be immediately subordinate to the URD. Thus, if the user volume name is SYS01 and the pathname given is >DIR1>FILEA, the full pathname becomes ^SYS01>DIR1>FILEA. If the pathname begins with two greater-than symbols, it is assumed to be directly subordinate to the SRD.

Relative Pathname

A relative pathname is one that begins with a file or directory name or a less-than (<) symbol. When a relative pathname begins with an element name, the first (or only) name in the pathname identifies a directory or file immediately subordinate to the working directory. When the relative pathname begins with one (or more) less-than symbols, the first (or only) name in the pathname identifies a directory or file immediately subordinate to the directory reached by moving from the working directory toward the root the number of levels indicated by the less-than symbol(s).

A relative pathname can consist of one or more elements. If a relative pathname contains more than one element, each element except the last must be a directory name, the first immediately subordinate to the current working directory level, the second immediately subordinate to the first, and so on. The last or only element can be either a directory name or a file name, depending on the function being performed, as described previously.

A simple pathname is a special case of the relative pathname. A simple pathname consists of only one element: the name of the desired entry in the working directory.

You can refer to a file or directory that is on the same volume but not subordinate to the working directory in two ways: by using an absolute pathname, or by using any of the forms of relative pathname previously described.

Figure 2-3 shows some relative pathnames and the full pathnames they represent when the working directory pathname is:

```
>PROJ1>USERA
```

DISK DEVICE PATHNAME CONSTRUCTION

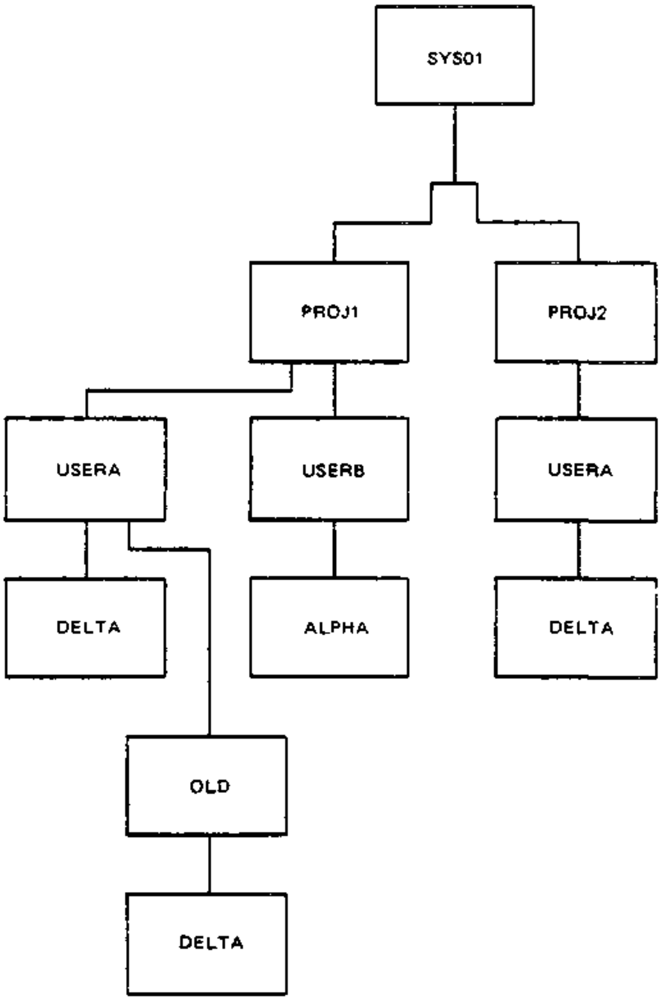
A special pathname convention is used to specify an entire disk volume (e.g., during a volume copy or volume dump). The special pathname consists of an exclamation point (!) followed by the symbolic device name and, optionally, the name of the disk volume. The general form of the disk device-level pathname is:

```
!dev_name[>vol_id]
```

where dev_name is the symbolic device name defined for the disk device at system building, and vol_id is the name of the disk volume.

If the vol_id is not supplied, reservation of the disk is exclusive (i.e., the reserving task group has read and write access but other users are not allowed to share the volume). This pathname form is used when creating a new volume. If the vol_id is specified, reservation is read/share (i.e., the reserving task group has read access only, other users may read and write). This pathname form is used when dumping selected portions of a volume without regard for the hierarchical File System tree structure.

RELATIVE PATHNAME[a]	FULL PATHNAME
DELTA	^SYS01>PROJ1>USERA>DELTA
OLD>DELTA	^SYS01>PROJ1>USERA>OLD>DELTA
<USERB>ALPHA	^SYS01>PROJ1>USERB>ALPHA
<<PROJ2>USERA>DELTA	^SYS01>PROJ2>USERA>DELTA
<	^SYS01>PROJ1



[a] Assume current working directory is ^SYS01>PROJ1>USERA

Figure 2-3. Sample Pathnames

Automatic Disk Volume Recognition

Automatic volume recognition dynamically notes the mounting of a disk volume. This feature allows the File System to record the volume identification and the root directory name in a device table. All references to disk files and directories begin, either explicitly or implicitly, with a root directory name; therefore, every mounted file is accessible to the File System software.

Disk File Organization

Since no one disk file organization can meet the needs of all users at all times, MOD 400 supports several different organizations, each of which is well suited to a particular application. Most of the supported organizations are based on the concept of a control interval (a unit of transfer between main memory and disk) and are referred to as Unified File Access System (UFAS) files. UFAS file organizations provide a common level of file processing compatibility across the GCOS Executives.

You establish the organization of a data file when you create the file with the Create File (CR) command. You read and write the file using statements and macro calls provided by the MOD 400 compilers and Assembler.

The following paragraphs summarize the MOD 400 disk file organizations. Refer to the GCOS 6 Data File Organizations and Formats manual (Order No. CZ19) for detailed descriptions of each organization.

UFAS SEQUENTIAL DISK FILE ORGANIZATION

Logical records are normally read from or written to a sequential file in consecutive order. Records must be written sequentially although the file can be positioned for writing through the use of a simple key. Records can be read, modified, or deleted directly when you specify their exact control interval and record address (simple key). Records cannot be inserted; they can be appended to the end of a file. Fixed- or variable-length records can be used. If a record is deleted, the position it occupied cannot be reused.

UFAS RELATIVE DISK FILE ORGANIZATION

A relative disk file can contain fixed- or variable-length records. If variable-length records are used, they occupy fixed-length slots (and the size of the largest record must be specified). Both sequential and direct access are supported; in direct access, simple and relative keys can be used. A record can be updated (i.e., rewritten), deleted, or appended to the file. If a record is deleted, the position it occupied can be used for a new record. A file can be created directly if you specify relative record numbers in random sequence.

UFAS INDEXED DISK FILE ORGANIZATION

Each logical record contains a fixed-size key field that occupies a fixed position. Records are logically ordered by key value; they can be accessed sequentially in key sequence or directly by key value. Fixed- or variable-length records can be used. Variable-length records are handled in variable-length format. A record can be updated, deleted, or inserted in key sequence into available free space. When no space is available to insert a record in key sequence, the record is placed in an overflow area. When the file is initially loaded, the records must be supplied in sequence by key value.

UFAS RANDOM DISK FILE ORGANIZATION

Records are accessed directly or sequentially. Variable-length records are handled in variable-length formats. Direct access of records is performed through CALC keys, which are fixed in size and located within each record. Records are positioned according to a technique involving an arithmetic derivation of their CALC keys; this derivation is called a hashing algorithm (and is carried out by the system). Insertions, updates, and deletions are handled according to key value.

UFAS DYNAMIC DISK FILE ORGANIZATION

A dynamic disk file can contain fixed- or variable-length records and supports inventory information to describe available space. The main purpose of this file organization is to provide an efficient storage organization for records to be accessed through alternate indexes.

Records are accessed sequentially or directly. Variable-length records are handled in variable-length formats. Records can be accessed indirectly through alternate indexes or directly by specifying their exact control interval and record address (simple key). Records are inserted into the file according to inventory information on a "best fit" basis.

Alternate Indexes

Alternate indexes allow you to define any number of alternate record keys to provide any number of different access paths to data records on disk. In effect, alternate indexes provide different views of the same data. The same data file can be viewed in many different ways by having more than one alternate index. For example, an application could have a relative file (ordered according to employee identification number) with alternate indexes for employee names and social security numbers. You could read such a file as a relative file ordered by employee numbers, or as an indexed file ordered by employee names, or as an indexed file ordered by social security numbers.

The alternate index capability exists in addition to the normal access mode based on type of file. You can establish an alternate index for any UFAS relative, indexed, random, or dynamic disk file. A file with more than one index can be accessed in a number of ways. The manner in which the file is reserved (through the Get File command) determines how the file is accessed. If the data file itself is reserved, the file can be accessed normally (i.e., based on file organization) or by a key that is supported by one of the indexes. When the data file is reserved through an alternate index, the contents of the file can be accessed as a standard indexed file. Additionally, if more than one index exists, the indexes can be used as alternate keys to refer to the data. When an alternate index is used for file reservation, that index is used as the primary key and the remaining indexes can be used as alternate keys. Any index can be selected as a primary index. When one index is used to access the file, it and the other indexes are automatically updated as the file is updated.

UFAS dynamic disk files contain inventory information to manage available file space. Therefore, in highly volatile file environments that include many insert and delete operations, dynamic disk files are the ideal data files to be used with alternate indexes.

Character string, signed binary, signed unpacked decimal, and signed or unsigned decimal key types can be used. Single component keys, ordered in ascending or descending sequence, are supported. Duplicate keys (more than one record in a file with the same key value) are supported on an index-by-index basis.

An alternate index is created with the Create Index (CX) command. Arguments of this command specify the name of the index and the name of the data file with which it is to be associated. The system creates the index on the same directory as the data file and with the same control interval size as that of the data file.

Refer to the Data File Organizations and Formats manual for further information.

Disk File Protection

The File System provides facilities that enable you to control the access to files and directories, to control the concurrent access to files, and to control the contention for records within shared files.

ACCESS CONTROL

Access control is an optional File System feature that allows the creator of a file or directory to specify which users (if any) are to be granted access to the file or directory and what types of access these users are to be granted.

There are two general forms of access control: Access Control Lists (ACLs) and Common Access Control Lists (CACLs). ACLs apply directly to a file or directory; CACLs apply equally to all immediately subordinate entries in a directory. You manage entries in the ACLs and CACLs by Set, Delete, and List Access commands.

Access control is a file or directory attribute. The File System maintains in each directory a list of users and the type of access each user is allowed. If a directory does not contain such a list, the items contained within it are not protected and are accessible to all users. (Access control applies only to disk files and directories. Tape files and other device-type files such as terminals, card readers, and paper tape readers, cannot be protected through the access control facility.)

Access Types

Access types for files are Read (R), Write (W), and Execute (E); access types for directories are List (L), Modify (M), and Create (C). A Null (N) access type applies to both files and directories; null access indicates that no access is to be granted.

Access Control/User_Id Relationship

The system builder can require that users of selected terminals log in to the system. Users who log in must be identified by a user_id consisting of three elements. Alternatively, the elements of a user_id can be entered as arguments in a Spawn Group, Enter Group Request, or Enter Batch Request command. These elements are:

```
        person.account.mode

     person - Name of individual who may access the system.

    account - Name of account to which work is charged.

       mode - Name of mode in which user is working (e.g.,
              interactive, batch, or operator).
```

The components are separated with periods (.). When comparing user_id's, any or all of the components can be replaced by an asterisk (*); for example:

```
        *.account.mode
        person.account.*
        *.*.*
```

When an asterisk appears in a component position, it is interpreted to mean any value that may exist. For example, if two persons (SMITH and JONES) are registered in an account named FILE_SYS, the user_id *.FILE_SYS.* matches either person in any possible mode. *.FILE_SYS.* matches all individuals registered to use FILE_SYS in any mode.

Access Control Lists

There are four kinds of access control lists: file ACLs, directory ACLs, file CACLs, and directory CACLs.

- File ACL: An ACL that applies to a specific file and is considered to be a file attribute. It contains a list of those users who can access the file and their specific access rights (i.e., read, write, execute).

- Directory ACL: An ACL that applies to a specific directory and is considered to be a directory attribute. It contains a list of those users who can access the directory and their specific access rights (i.e., list, modify, create).

- File CACL: A CACL that applies to all files immediately subordinate to a directory. A file CACL is considered to be a directory attribute that applies only to files contained in that directory. A file CACL contains a list of file users and their specific access rights (i.e., read, write, execute). Use of file CACLs can save disk space and search time if all or most files in a directory have the same access requirements. A file CACL does not override individual file ACLs set on files in the directory.

- Directory CACL: A CACL that applies to all directories immediately subordinate to a directory. A directory CACL is considered to be a directory attribute that applies only to immediately subordinate directories. A directory CACL contains a list of directory users and their specific access rights (i.e., list, modify, create). Use of directory CACLs can save disk space and search time when all or most subdirectories have the same access requirements. A directory CACL does not override individual directory ACLs set on the subdirectories.

Checking Access Rights

When you reserve a file (through the $GTFIL system service macro call or the Get command), the File System checks your right to access that file. You are said to be on the access control list if your user_id matches an entry on the ACL or CACL in any of the forms noted below.

Universal access (no access restriction) is implied if neither an ACL nor a CACL exists for the file being reserved. If either list is present, it is scanned by access control.

The checking priority is ACL first, CACL second. If a match is found in the ACL for a fully specified user_id (all three components explicitly stated), the CACL is not inspected. If a match is found on a partially specified user_id (one or more components specified as an asterisk), the CACL is inspected for a more explicitly stated user_id. The following list indicates the priority hierarchy of user_id formats in order of decreasing priority.

1. person.account.mode
2. person.account.*
3. person.*.mode
4. person.*.*
5. *.account.mode
6. *.account.*
7. *.*.mode
8. *.*.*

Access is checked only for the target file or directory; the access rights set on directories that may be traversed in reaching the target file are not checked. You may be denied access at some intermediate directory level and still gain access to a subordinate directory or file.

FILE CONCURRENCY CONTROL

Concurrent read or write use of a file is established by the task group that first reserves the file. Concurrency has two aspects: (1) it establishes how tasks in the reserving task group intend to access the file, and (2) it establishes what the reserving task group allows other task groups to do with a file. If the file is already reserved, a task group's concurrency request is denied when its intended access conflicts with the access permitted by another task group. The concurrency request is also denied if what it allows others to do conflicts with the access already established by another task group. For example, if a task group reserves the file exclusively, other task groups are denied any access. Or, if a task group permits read-only access but does not permit write access, other readers are allowed but writers are denied access.

Concurrency is controlled through the Get command or through the $GTFIL system service macro call. The possible combinations of access intended for the reserving task group and the sharability permitted other task groups are given in Table 2-1.

Table 2-1. Disk File Concurrency Control

Reserving Task Group	Other Task Groups
Read only	Read only (Read share) Read or Write (Read/Write share)
Read or Write	No Read, no Write (Exclusive use) Read only (Read share) Read or Write (Read/write share)

Compiler-generated programs, commands, sort operations, and other system software always request exclusive concurrency for files that they reserve for a user. The operator terminal must be reserved with read/write shared concurrency to allow concurrent access by many task groups. For this reason, the command argument -COUT specifying the list output file cannot be the operator terminal. If the command-in and user-in files are on disk, they are reserved with read-only shared concurrency; if assigned to a user terminal, they are reserved with exclusive concurrency. The user-out and error-out files are always reserved for exclusive use.

ACCESS CONTROL/CONCURRENCY CONTROL RELATIONSHIP

In an environment that employs access control, users must have certain minimum types of access privilege to obtain the specific type of concurrency control they indicate in their Get File commands or $GTFIL system service macro calls.

Table 2-2 summarizes the relationship between access control and concurrency control for disk files, disk directories, and disk volumes. (Note that access control does not exist for other types of devices.)

Table 2-2. Access Control/Concurrency Control Relationship

Object	Desired Concurrency	Minimum Access
Disk Files	Read Read/Write	Read Read/Write
Disk Directories	Exclusive Use Nonexclusive Use	List/Modify List
Disk Volumes	Read or Read/Write	Modify access to root directory

SHARED FILE PROTECTION (RECORD LOCKING)

Record locking can be set as a file attribute (when the file is created or modified) or temporarily set at each file reservation. In MOD 400 record locking is performed when you read or write a record. It involves locking the control interval containing the record, making it inaccessible to other users until it is explicitly unlocked. Record locking is performed on a shared-read, exclusive-write basis.

- Shared-Read: If you are the first user who attempts to access a given record and you are a "reader" (a declaration of intent made at OPEN time), then other concurrent readers of the file can also read the record, but it is locked to any writers.

- Exclusive-Write: If you are the first user who attempts to access a given record and you are a "writer" (as declared at OPEN time), then, regardless of whether this access is a read or a write, the record is locked to all other concurrent users.

Even though a file is reserved or created with record locking specified, a special file reservation option allows you to bypass record locking for read-only access. Simply specify read-only access with read/write sharing by others. Note that in this case the integrity of the data being read is not guaranteed.

Locked records are released upon explicit request (through a $CLPNT macro call), when the file is closed, or when the task group terminates.

MULTIVOLUME DISK FILES

In most applications a disk file resides on a single volume. However, there may be situations in which you want to extend a file over more than one physical volume. The need for multivolume files could arise from any of the following:

- You want to have an endless sequential file capability similar to that available with magnetic tape.

- You want to define a single file that is too large to be contained on one volume.

- You want to improve access time to a file by spreading the file data over several volumes, and/or separating the index portion of an indexed file from the data portion and placing the portions on separate volumes.

In producing a multivolume file, you treat your disk file as a collection of file sections. A file section is that part of the file that is contained on one volume. A file set is all of the sections making up the multivolume file.

Multivolume Sets

A multivolume set is a disk file that resides on more than one volume. A volume is identified as being part of a multivolume set when the volume is created using the Create Volume (CV) command.

Each multivolume set has a root volume and a number of additional volumes. All volumes that are part of the set are called members. The root volume is always member number 1.

The name of a multivolume set is independent of the names of the volumes it contains. (The set name has the same format as the volume name.) You establish that a volume is a member of a set by specifying the set name and a sequential member number at volume creation.

There are two types of multivolume sets: online and serial. Online multivolume sets are used for files that must extend over several volumes because they are too large for one volume or because you want to improve the access time to the files by having the data (or data and index) spread over multiple volumes. Serial multivolume sets are used for files that require a sequential capability similar to that of magnetic tape.

When you create a volume, you specify whether the volume is part of an online multivolume set or part of a serial multivolume set. You designate an online multivolume file by creating it under a directory in the root volume of an online multivolume set. You designate a serial multivolume file by creating it under the root directory of the volume in the serial multivolume set on which the file is to start.

ONLINE MULTIVOLUME SET

An online multivolume set has the following characteristics:

- All members of the set must be mounted and available when the set is in use.
- Member volumes, other than the root volume, can be used independently of other members in the set to contain single-volume files and directories.

ONLINE MULTIVOLUME FILE

A file is established as an online multivolume file when it is created in some directory on the root volume of an online multivolume set. An online multivolume file has the following characteristics:

- Can have any UFAS file organization
- Can be located by any type of pathname
- Need not continue serially from one volume to the next.

Figure 2-4 illustrates the combination of files and volumes used by a sample online multivolume set. Multivolume files FILEA, FILEB, and FILEC must begin on VOL1. FILEX, FILEY, and FILEZ are single-volume files. The pathnames used to access the files are shown at the bottom of the figure.

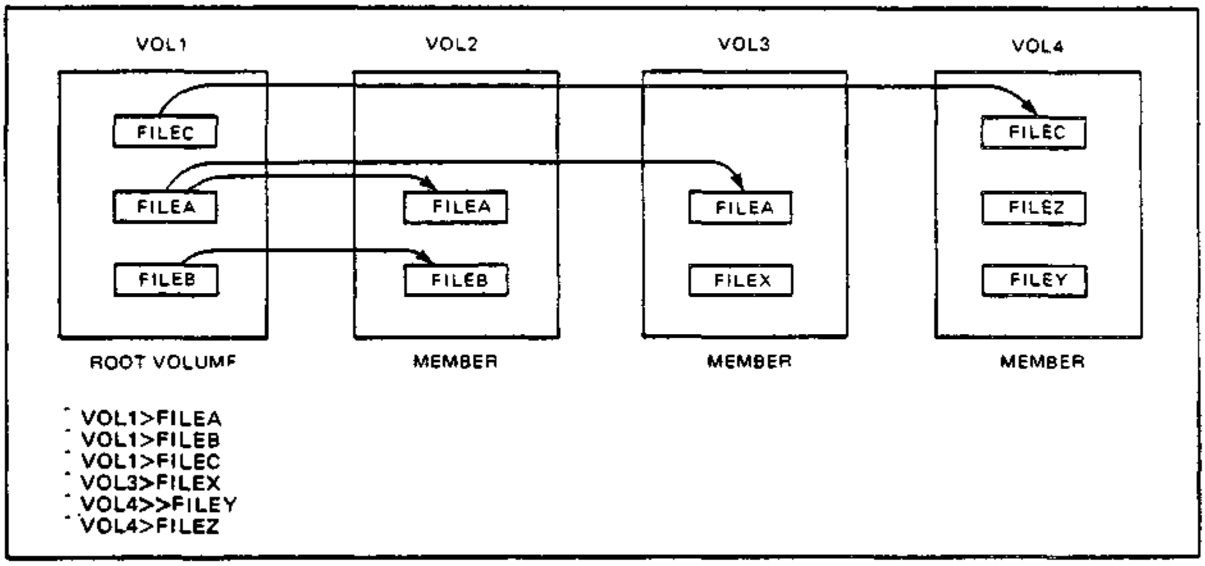


Figure 2-4. Example of Online Multivolume Files

SERIAL MULTIVOLUME SET

A serial multivolume set has the following characteristics:

- No member of the set need be mounted until it is required for processing.
- Any member of the set, including the root volume, can be used independently of other members of the set to contain single-volume files and directories.

SERIAL MULTIVOLUME FILE

A file is established as a serial multivolume file when it is created in the root directory of a volume in the serial multivolume set. A serial multivolume file has the following characteristics:

- Must be a UFAS sequential file
- Must be cataloged in the root directory of the volume on which it starts; more than one serial multivolume file can belong to a set (each such file can begin on a different volume if desired)

- Must be located through a pathname of the form: ^volid>filename

- Must continue serially from one volume to the next.

Figure 2-5 illustrates the combination of files and volumes used in a sample serial multivolume set. Serial multivolume file A begins in VOL1. Serial multivolume file B begins in VOL2. Both continue in other volumes of the set. Files C, D, and E are single-volume files. The pathnames by which the files are located are shown at the bottom of the figure.

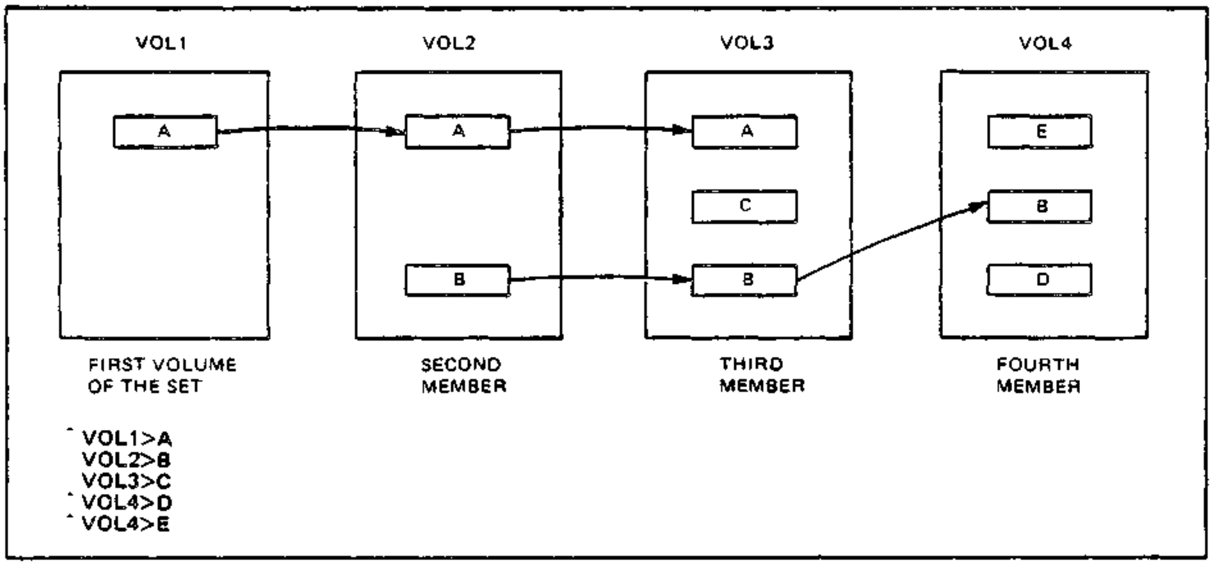


Figure 2-5. Example of Serial Multivolume Files

Multivolume Disk File Overhead Requirements

Serial multivolume disk files require no more overhead than single-volume disk files. The extent field information for online multivolume disk files is contained in the file directory. (An extent is a group of contiguous allocated disk sectors.) An indirect extent holds the relative volume number that contains the succeeding set of extents. The first extent for each online multivolume file is an indirect extent if the first set of extents is not located on the root volume. Each time the volume changes, an indirect extent appears.

MAGNETIC TAPE FILE CONVENTIONS

The magnetic tape file conventions discussed in the following paragraphs include tape file organization, tape file naming conventions, and tape file pathnames.

Tape File Organization

This information applies only to 9-track magnetic tape.

Magnetic tape supports only the sequential file organization. Fixed- or variable-length records can be used. Records cannot be inserted, deleted, or modified, but they can be appended to the end of the file. The tape can be positioned forward or backward any number of records.

The unit of transfer between memory and a tape file is a block. Block size varies depending on the number of records and whether the records are fixed or variable in length.

A block can be treated as one logical record called an "undefined" record. An undefined record is read or written without being blocked, unblocked, or otherwise altered by data management. Spanned records (i.e., those that span across two or more blocks) are supported. (No record positioning is allowed with spanned records.)

A labeled tape is one that conforms to the current tape standard for volume and file labels issued by the American National Standard Institute (ANSI). The following types of labeled tapes are supported:

- Single-volume, single-file
- Multivolume, single-file
- Single-volume, multifile
- Multivolume, multifile.

The following types of unlabeled tapes are supported:

- Single-volume, single-file
- Single-volume, multifile.

Magnetic Tape File and Volume Names

Each tape file and volume name in the File System can consist of the following ASCII characters:

digits (0 through 9)
uppercase alphabetics (A through Z)
! (exclamation point)
" (double quotation marks)
$ (dollar sign)
% (percent sign)
& (ampersand)
' (apostrophe)
((left parenthesis)
) (right parenthesis)
* (asterisk)
+ (plus sign)
, (comma)

- (hyphen)
. (period)
/ (slash)
: (colon)
; (semicolon)
< (less-than sign)
= (equal sign)
? (question mark)
_ (underscore)

The underscore character (_) can be used as a substitute for a space. If a lowercase alphabetic character is used, it is converted to its uppercase counterpart.

Any of the above characters can be used as the first character of a file or volume name.

The name of a tape volume can be from 1 through 6 characters in length; tape file names can be from 1 through 17 characters.

Magnetic Tape Device Pathname Construction

A magnetic tape volume must be dedicated to a single user. For this reason, the device pathname convention must always be used when referring to magnetic tape volumes or files. The general form of a tape device file pathname is:

```
!dev_name[>vol_id[>filename]]
```

where dev_name is the symbolic name defined for the tape device at system building, vol_id is the name of the tape volume, and filename is the name of the file on the volume. Tape devices are always reserved for exclusive use (i.e., the reserving task group has read and write access; other users are not allowed to share the file).

Automatic Tape Volume Recognition

Automatic volume recognition dynamically notes the mounting of a tape volume. This feature allows the File System to record the volume identification in a device table, thus making every tape volume accessible to the File System software.

UNIT RECORD DEVICE FILE CONVENTIONS

Unit record devices (e.g., card readers, card punches, printers, paper tape readers, and paper tape reader/punches) are used only for reading and writing data; they are not used for data storage and thus do not require conventions for file identification and location.

Refer to a unit record device by entering a pathname consisting of an exclamation point (!), followed by the symbolic device name defined during system building. The format is as follows:

```
!dev_name
```

where dev_name is the symbolic device name of the unit record device.

FILE SYSTEM BUFFERING OPERATIONS

A buffer is a storage area used to compensate for a difference in the rate of data flow, or time of occurrence of events, during transmission of data from one device to another. As used in I/O programming, the term "buffer" refers to an I/O area in systems that provide the possibility of I/O overlap. Buffering is the process of allocating and scheduling the use of buffers. In sequential data processing, for example, overlap of input operations and processing can be achieved by anticipatory buffering where the next block is read into memory before it is needed. The program can then process records from block n while block n+1 is read into memory.

MOD 400 supports different types of buffered operations for disk devices, magnetic tape devices, and unit record and terminal devices.

Disk Buffered Operations (Buffer Pools)

Disk files can be accessed at the block or record level. In block level access, data is transferred directly between the file and a buffer in the user program; the user program must perform all buffer management operations. In record level access, the system assigns disk files to buffer pools when the user program opens the files; the system buffering facilities are used to perform all buffer management operations.

When you open a file for read, write, rewrite, and delete operations, the File System assigns that file to a particular buffer pool. All buffers in a pool are the same size. Any number of files with matching control interval sizes can be assigned to the same buffer pool; however, a particular file can be assigned to only one pool.

Each buffer in a buffer pool contains a disk control interval. When an application program issues a read instruction and the desired record is not in any buffer, the next empty buffer is filled with the control interval containing the record. When all buffers are filled, an active buffer is selected for the next different control interval according to a least-recent-usage algorithm.

Buffer pools conserve memory when disk files are accessed, and they eliminate the need for each user to define his/her own buffer areas. One or more system-wide buffer pools should be created at system startup (through a startup EC file). Users who have special buffering requirements can create their own buffer pools for files they reserve exclusively.

The paragraphs below describe the types of buffer pools and the way in which files are assigned to the pools.

TYPES OF BUFFER POOLS

Each buffer pool is created as either a public or private buffer pool and can be considered file-specific or general. Buffer pools are created by the Create Buffer Pool (CBP) command and are deleted by the Delete Buffer Pool (DBP) command. When you create a buffer pool, specify its name (this is optional), the number of buffers it is to contain, and the size of each buffer.

Public Buffer Pools

Public buffer pools are those created by the operator or the system startup EC file. Public buffer pools reside in system memory and are available to all files and task groups. A disk file is assigned to a public pool if its control interval size (specified in the CR command that created the file) matches the pool's buffer size.

In many environments, three or four public buffer pools corresponding to the three or four different file control interval sizes is sufficient for all performance/buffering needs.

Private Buffer Pools

Private buffer pools can be created by each user. Private buffer pools reside in the task group's memory space and are available only for disk files reserved exclusively by that task group. A disk file is assigned to a private pool if the file is reserved for exclusive use and its control interval size (specified in the CR command that created the file) matches the pool's buffer size. Private buffer pools should be created only if necessary to meet specific buffering needs. Public buffer pools should be sufficient in most cases.

File-Specific Buffer Pools

When you reserve a disk file with the Get File (GET) command, you can specify the number of buffers (using the -NBF argument) to be used when accessing the file. When the file is opened, a buffer pool is automatically created for use only by that file. This file-specific pool is created in the task group's memory if the file is reserved exclusively, or in system memory if the file is reserved as sharable. The -NBF argument of the CR command should be used carefully since it prevents a file from being assigned to a public or private buffer pool.

BUFFER POOL STATISTICS

The File System collects a set of statistics on the use of each buffer pool. The installation can use this information to optimize disk I/O operations. The statistics are obtained through the Buffer Pool Status (BPS) and Buffer Pool Information (BPI) commands. The BPS command provides a summary of the public or private buffer pool status. The BPI command provides a detailed status report on a particular buffer pool.

Magnetic Tape Buffered Operations

The -NBF argument of the Get File (GET) command can be used with magnetic tape files to reserve one or two buffers. If -NBF is not specified, the File System attempts to allocate two buffers. If two buffers are allocated, the File System does "double buffering." When the tape file is being read, the File System unblocks one buffer while an anticipatory read is done into the other buffer. Similarly, when the tape file is being written, the File System blocks records into one buffer while a previously filled block is written out of the other buffer. This allows application code to execute in parallel with I/O transfers.

Unit Record and Terminal Buffered Operations

All printers and most interactive terminals are provided with one File System buffer. (The operator terminal (LRN 0) cannot be buffered.) By providing a File System buffer, asynchronous I/O can be done; that is, application code can execute in parallel with I/O transfers.

All terminals (except the operator's) and printers have tabbing capability through software that converts the tab into spaces. Default tabulation stops are set at position 11 and every 10th position thereafter for the line length of the device.

BUFFERED READ OPERATIONS

An application task issues a logical READ to a File System buffered device. If the buffer is full from a prior anticipatory read, the data in the buffer is transferred into the application task's area; then a physical I/O transfer into the system buffer (an anticipatory read) is performed in parallel with continued task execution. If the buffer is not full, task execution stalls until the anticipatory read is completed.

The timing of the initial anticipatory read performed for the card reader is different from that of the interactive terminals; for other read actions it is the same. An application task issues an OPEN call to the card reader. Immediately after the OPEN is complete, the File System performs an asynchronous anticipatory read into the system buffer while the application continues execution. All OPEN calls are synchronous.

For interactive terminals, immediately after the OPEN is complete an asynchronous physical connect is performed while the application continues execution. Assembly or FORTRAN applications can check the status of the OPEN to see if a READ can be issued without stalling application execution. The File System issues an asynchronous anticipatory physical read when a status check after the physical connect is complete. The file status remains busy until the physical read is done and the system buffer is full. At this point, the file status is "not busy" (i.e., the anticipatory read is successfully completed), and the application can issue a READ with the assurance of receiving data immediately. If at any point after the OPEN is issued the Assembly or FORTRAN application issues a READ before the physical connect and anticipatory read have been completed, the READ is synchronous and further central processor execution is stalled on this application until the anticipatory read is complete. To avoid status check looping to test the input buffer status or stalling on a READ, both Assembly and FORTRAN applications can put themselves into the Wait state, thus making the central processor available for lower priority tasks. After the OPEN, a COBOL application must issue READ requests. The COBOL application will be put in the Wait state if it is executing its I/O statements in synchronous mode. Otherwise, the COBOL run-time package performs the status checks and returns a 9I status until successful completion. The COBOL program can either loop on the READ or continue other processing.

The anticipatory read allows an application to control input from more than one interactive terminal, each of which represents a data entry terminal. By testing the status of the system buffer before a READ (FORTRAN, Assembly) or by checking for the 9I status return after a READ (COBOL) even if a terminal operator is not present at the time of the READ request, the application will not be stalled and it can continue to poll other terminals.

BUFFERED WRITE OPERATIONS

A buffered write operation to a device works on behalf of the application program in the same logical manner as the read: the program is permitted to execute in parallel with the physical I/O transfer to the device. To achieve this parallel processing, no special operation occurs on an OPEN call and no distinction is made between interactive and noninteractive file types. Each WRITE call is completed by moving data from the application buffer to the File System's buffer (performing any detabbing, if requested), initiating the transfer, and returning control to the application program. If the program performs a second WRITE while the system buffer is still in use for a previous transfer, the application is stalled until the buffer is available and new data moved into it again. The application can avoid stalling execution by checking the status of the system buffer before issuing a WRITE to an interactive terminal to see if, in a special mode, it is still in use or not (FORTRAN, Assembly) or by testing for the 9I status return after the WRITE (COBOL, for interactive devices only).

If a WRITE call is issued while data is being entered (because of a read) into the system buffer, the read is allowed to complete and input data is saved in the system buffer, a synchronous write is reissued by File System, and output data is transferred directly from the application buffer. However, tab characters are not expanded into spaces by software.

Special considerations for buffered write operations arise because, if a physical I/O error occurs while data is being transferred from the system buffer to the device, the application program must be aware that the error occurred on the previous write operation. Furthermore, if any error does occur, the application program may need to have saved (or be able to retrieve) the data record so that it can be repeated.

Section 3
SYSTEM ACCESS

SYSTEM CONFIGURATION AND ENVIRONMENT DEFINITION

At larger installations, a system programmer or administrator might design the configuration files and the different operating environments for his/her installation. The daily startup would be done by an operator. At smaller installations, especially those where programmers run dedicated applications, each programmer might do the configuration and startup for his/her own application.

Creation of a usable system consists of a two-step procedure:

1. Bootstrap a Honeywell-supplied system startup routine that provides a limited operating environment for building the files used in the second step.

2. Specialize the system startup procedure by configuring a system to correspond to the installed hardware and by defining the environment in which to prepare and execute applications programs.

The bootstrap operation simply consists of turning on the power supply to the hardware, mounting the disks containing the MOD 400 System software, and pressing several control panel keys (including Bootstrap Load) to execute a standard bootstrap routine. This procedure is described in the GCOS 6 MOD 400 System User's Guide (Order No. CZ04). The bootstrap operation generates the initial configuration and startup operations. Procedures are executed to provide a one-user online environment that can be used to specialize system startup, perform program preparation, or perform application program execution.

In the limited environment created as a result of the bootstrap procedure, the user must build a file of directives that describes the operating environment that will exist at the installation. This file of directives (called a CLM_USER file) is easily created using an interactive system definition program called M4_SYSDEF. M4_SYSDEF conducts an interactive dialogue with the system builder; M4_SYSDEF then creates a CLM_USER file based on information supplied by the builder during the interactive session. If necessary, the CLM_USER file generated by M4_SYSDEF can be modified using the line editor. Alternatively, you can hand-build the entire CLM_USER file using the line editor. Both methods are detailed in the System Building and Administration manual.

Additionally, to define the desired environment further, you can modify the vendor-supplied START_UP.EC file of operator commands that is immediately subordinate to the root directory. (START_UP.EC files are described later in this section.)

After these files are created, the system is again bootstrapped. This time, however, directives in the CLM_USER file control the configuration, and the operator commands in the START_UP.EC file define the operating environment.

USER REGISTRATION

User registration is a process that protects the system from unauthorized access. Once you configure user registration, you cannot deconfigure it.

Using the interactive dialog invoked by the Edit Profile (EP) command, the system administrator creates and modifies the profiles file. The profiles file is the file that contains all the user profiles for registered users.

User profiles are made up of sections. A user profile always contains a Registration (RE) section. The defaults section login line defaults, login id, encrypted password, login traits, date/time of current login, and the count of instances of invalid password use.

A user profile can contain the following optional sections:

- Statistics Section: Contains date and time of last login and logoff, elapsed time of last session, total time of all sessions, number of sessions, name of last terminal used, and date of initial registration.

- Comments Section: Contains any comments the system administrator has about this user's registration.

- Subsystem Sections (one or more): Contains user-specific subsystem information meaningful only to the individual subsystems.

The Listener uses the user profile to monitor the privileges and/or limitations of each user.

The List Profile (LP) command allows registered system users to view the contents of their personal user profiles and allows a system administrator to view any registered user's profile.

Refer to the System Building and Administration manual for details on user registration, EP, and LP.

ACCESSING THE SYSTEM

The following paragraphs describe the methods of accessing the system.

Ways to Access the System

An installation can simultaneously support several ways to access a system, so you must determine the access available at your terminal. For simplicity of discussion, three types of access will be described. Access can be through a Login command, operator control, or your own application design.

LOGGING IN

At system building time, the Listener component can be configured to monitor a set of designated terminals. Once the Listener has been activated by the operator, it determines which terminals to monitor for system access from information in a user-created terminals file. The terminals file is made up of G-, T-, and A-records:

- G-record: There is one G-record in the terminals file. This record specifies the number of concurrently logged in users that will be allowed on the system.

- T-record: Each terminal that is to be used as a login terminal must be identified in a T-record that gives the symbolic device name (sympd) of the terminal. The record for a direct login terminal must also contain the command line image to be used when the terminal is turned on.

- A-record: Each abbreviation for a Login command must be identified in an A-record that specifies the abbreviation and the associated Login command line image to be used when the abbreviation is entered from a terminal.

Refer to the System Building and Administration manual for details about constructing a terminals file and activating the Listener.

You can gain access to the system through a Listener-monitored terminal with either a Login command line or a combination Login command line and password. You can log in by doing one of the following:

1. Type in a Login command as described in the GCOS 6 MOD 400 Commands manual (Order No. CZ17) and then, if required, type in your designated password.

2. Type in the abbreviation of a specific Login command line and then, if required, type in your designated password.

3. Turn on the terminal and be logged in through a direct login.

OPERATOR ASSIGNED ACCESS

An application can be activated with a terminal designated for input of its commands or user input required by the program executing the command. Terminals that are used for logging in cannot be assigned to an application. An installation can have a mixture of terminals: some that may be used for logging in and others that may be assigned directly to applications or another user.

USER DESIGNED ACCESS

A user at an installation that allows use of the system for a single dedicated application must:

- Configure and startup the system
- Act as the operator
- Determine what the application environment should be
- Decide how to access the system for that application.

The same terminal can be used both as an operator terminal and user terminal (i.e., a dual-purpose operator terminal), as described in the System User's Guide.

Activated Lead Task

When you successfully gain access to the system, the executable code for the lead task (i.e., the controlling task of the application) is loaded and activated. The lead task can be designated to be either the Command Processor or an application. When the Command Processor is the lead task, you have complete flexibility to control execution by being able to execute any user command in the Commands manual. When an application is the lead task, the Command Processor is not part of the task group.

COMMAND ENVIRONMENT

The command environment is that environment in which you communicate with the Executive through the use of the command lines entered at a terminal, read from a command file or through the User Productivity Facility (UPF).

User Productivity Facility

The UPF provides an easy-to-use interface to MOD 400. Instead of typing command lines, you use online menus and forms. You select system commands by choosing options listed on menus and then fill in fields on one or more forms. When you have filled in all appropriate fields on the forms, the command is executed. Menu selections are grouped by function types and by use.

Each selection on a menu and each field on a form that must be filled in has an associated help message. If you need help in selecting an option on a menu or in filling in a field on a form, press the key designated as the help key for your terminal. A three-line help message is displayed below the menu or form. The help message tells you what to do next. If you make a mistake when you fill in a field, the system provides expanded error messages. These expanded messages list causes and corrective actions to help resolve the problem.

You can modify the UPF menus, forms, help messages, and error messages using the same tools that Honeywell used to create them. You can add, delete, modify, and reword any of the UPF elements using the following tools:

- Menu Builder to create, modify, or delete menus
- Forms Developer (VFORMS) to create, modify, or delete forms
- Generalized Forms Processor to process standard forms
- Add Delete Message Utlity to build or change a message library.

For detailed information about using and maintaining UPF, refer to the GCOS 6 MOD 400 Menu Management/Maintenance Guide (Order No. CZ10).

Command Processor

The essential parts of the command environment, from your point of view, are the Command Processor and the command-in file. The Command Processor is the system software component that reads your command lines. It interprets them into procedures that load and initiate execution of bound units, which fulfill the requests represented by the command lines. The command-in file is the file from which the command lines are read. It can be a terminal device, as in the case of an interactive user, or a command file stored on disk or on cards, as in the case of a noninteractive user.

Three other files are involved with, but not limited to, the command environment. These are the user-in file, the user-out file, and the error-out file.

USER-IN FILE

The user-in file is the file from which a command, during its execution, reads it own input. When a task group request has been processed, and as long as no alternate user-in file is specifed as an argument in a subsequent command, the user-in file remains the same as the command-in file. At the termination of a command that names an alternate user-in file, the user-in file reverts to its initial assignment.

The directives submitted to the line editor following the entry of the line editor command, for example, are submitted through user-in. No specific action is required on your part to activate, or to connect to, user-in unless the directives are to be read from a previously created disk file. You simply invoke the line editor and begin entering line editor directives through the same terminal; the attaching of the terminal to the user-in file is invisible to you.

USER-OUT FILE

The user-out file is the file to which a task group normally writes its output. However, certain system components (compilers, etc.) also write to list files (path.L) or to the output file defined in the -COUT argument. The user-out file is initially established by the -OUT argument of the Enter Batch Request (EBR), Enter Group Request (EGR), or Spawn Group (SG) commands. (Thus, originally, it is the same device as the error-out file device.) It can be reassigned to another device by use of the File Out (FO) command or by the use of the New User Out ($NUOUT) system service macro call. Such a reassignment remains in effect for the task group until another reassignment occurs.

Again using the line editor as an example, any responses from the line editor, such as the printing of a line of the file being edited, are issued through user-out. As in the case of user-in, you need not perform any special action to attach your terminal to the user-out file. The only time such action would be required is if you wanted to direct the output from the command to some device other than the terminal.

ERROR-OUT FILE

The error-out file is used by system to communicate to you an error condition that may be detected during the interpretation of a command or its subsequent execution. Such a condition could be a missing command argument, reported by the command processor, or a file-not-found condition, reported by the invoked command. The error-out file is the same as the initial user-out file. You cannot reassign error-out.

COMMAND LEVEL

The rest of this section describes in detail the functions available to you at the command level. When the system is in a state capable of accepting a command from command-in, it is said to be at command level.

Achieving Command Level

You can achieve command level by creating or spawning a user task group whose lead task is the Command Processor. (See the Commands manual for details on the Create Group and Spawn Group commands.) The Listener can also spawn task groups whose lead task is the Command Processor. (See the System Building and Administration manual for details on the Listener.) The system is delivered with a Honeywell-supplied task group ($H) whose lead task is the Command Processor.

Regardless of the way in which the system arrives at this state, the system indicates that it is at command level by issuing a "ready" prompter message at the user terminal. This assumes that you have not disabled the ready message by issuing a Ready Off (RDF) command; if you have, the system still comes to command level but you are not informed. Note that if you are working in the system task group ($S) at the operator terminal, no ready prompt message appears to inform you that you are at command level, unless you issue a Ready On (RDN) after an EC !CONSOLE command (see the System User's Guide for details).

When executing a command function, you can return to command level in one of two ways:

1. At normal termination of a command function, the task group returns to command level and awaits the entry of another command. It is not recommended that you enter the Bye (BYE) command if your terminal is not monitored by the Listener.

2. You can interrupt the execution of an invoked command by pressing the Break key (this key may be labeled with BREAK or BRK) on your terminal. The system then responds with the break message **BREAK**. At this point you can enter other commands, or the Start command to resume processing where it was interrupted. (See the Commands manual or the System User's Guide for details.)

Functions Performed at Command Level

When a command such as Copy (CP), Change Working Directory (CWD), or EGR is read by the Command Processor, the system spawns a task whose objective is to fulfill the requirements of the command. This action consists of the following steps:

1. A task is spawned naming the requested bound unit (i.e., command name). Task spawning implies task creation (i.e., the allocation and initialization by the system of any control structures and data areas required for task control.)

2. The Loader is called to load the requested bound unit.

3. A request for the bound unit's execution is placed against the created task and the Command Processor enters the Wait state to await completion of the requested task (command). At this point the system leaves command level, which can be returned to only by completion of execution of the command or by pressing the Break key on the terminal, as described previously.

4. If the command is EGR, it places a group request against an application task group and the EGR command terminates. The request is queued if there are other outstanding requests against the application task group from previous EGRs.

5. When the command terminates, the spawned task is deleted and a ready message is optionally issued to indicate that the system has returned to command level and can accept further commands.

COMMAND LINE FORMAT

Commands are read and interpreted by the Command Processor, which executes as the lead task in the batch task group or as the lead task in an online task group. Each command spawns a task that performs the requested function (e.g., create a task within an existing group, enter a group request, dump a file). When the execution of a command terminates, control is returned to the Command Processor, which can then accept another command.

A command line to the Processor is a string of up to 127 characters in the form:

```
command_name_1[arg_1...arg_n][;command_name_2[arg_1...arg_m]]...
```

where command_name_1 is the pathname of the bound unit that performs the command function. Each subsequent arg entry is an argument whose functions are described in the following sections. A command line can span one or more physical lines. A line is concatenated with the next line by ending it with an ampersand (&). A command line consisting of two or more concatenated lines can be canceled by entering a single ampersand on the next physical line. More than one command can be included in a command line by ending each command with a semicolon (;).

Arguments

An argument of a command is an individual item of data passed to the task of the named command. Some commands require no arguments; others accept one or more arguments as indicated in the syntax of each command description. The types of arguments used are:

- Positional Argument: An argument whose position in the command line indicates to which variable the item of data is applied. The argument can occur in a command line immediately after the command name or as the last argument following the control arguments, as in the List Names (LS) command.

- Control Argument: A keyword whose value specifies a command option. A keyword is a fixed-form character string preceded by a hyphen (e.g., -ECL). It can be alone, as in -WAIT, or it can be followed by a value, as in -FROM xx.

Except for -ARG or when the last argument of a command line is a positional argument, keywords of control arguments can be entered in any order in the line, following the initial positional arguments. The keyword -ARG must be the last argument of the SG, EBR, Enter Task Request (ETR), or Spawn Task (ST) command line. The arguments following -ARG are passed to activated (application) task.

Spaces in Command Lines

Arguments in command lines are separated from each other by spaces. Unless otherwise indicated, a space in a command line syntax represents one or more space characters, or one or more horizontal tab characters, or a combination of these. Spaces can be embedded within an argument by enclosing the argument in apostrophes (') or quotation mark (") characters. Note that a file name supplied in an argument can be shown to have a trailing space if the argument is bounded by quotation marks.

Parameters

Arguments are the user-selected items of data passed to a task. In the activated task, which is written in a generalized manner to handle any set of data passed to it, these data items are known as parameters. If the activated task expects positional parameters, the order of the command line arguments passed to it must be in the same order as the task's positional parameters.

Protected Strings

Special significance is attached to the following reserved characters:

- Space (Blank)
- Horizontal Tab
- Quotatation Mark (")
- Apostrophe (')
- Semicolon (;)
- Ampersand (&)
- Vertical Bar (|)
- Left Bracket and Right Bracket ([]) (active function delimiters)

It is occasionally necessary to use a reserved character without its special meaning (e.g., a blank could be used in a command argument). The protected string designators (the apostrophe and quotation mark) are reserved for this purpose. Reserved characters within a protected string (one surrounded by protected string designators) are treated as ordinary characters. Thus, in an argument:

```
-ARG "ALPHA 2" ALPHA
```

the space in ALPHA 2 is treated as part of the name.

Another example is the & followed by a number in a command-in file. If &1 does not represent a substitutable parameter, it must be written as &'1' or &"1" (not "&1"). Substitutable parameters are discussed in the Commands manual and in the System User's Guide.

Also, since protected string designators themselves are reserved characters, it may be desirable to suppress their special meaning. For this purpose, two adjacent protected string designators of the same type within a protected string of that type are treated as a single occurrence of that character. A protected string designator within a string enclosed by the complementary protected string designator is treated as an ordinary character. The following arguments:

```
-ARG "A""B"

-ARG 'A''B'
```

result in the strings A"B and A'B being passed to a command.

Active Strings and Active Functions

An active string is part of a command and is evaluated (executed) immediately by the Command Processor. The resulting value is then substituted for the active string characters in the command line.

Active functions are commands explicitly designed for use in active strings. The function in the string is evaluated and the resulting value is substituted in the command line for those active string characters that called the function. For example, the function [EQUAL a b] returns TRUE if a=b, and returns FALSE if a≠b. Active strings and active functions are described in the Commands manual and in the System User's Guide.

EC AND START_UP.EC FILES

The Command Processor is able to read commands from a source other than an interactive user terminal. Such a source can be in the form of an EC file. An EC file is one which you construct (using a text editor) that is destined to be read by the Command Processor invoked either by the Execute (EC) command or when a task group is activated with the Command Processor as its lead task and the EC file is specified as the task group's user-in file. The EC file might contain a series of commands that you execute on a frequent basis, such as commands to execute a set of applications program that run at the end of the month to summarize inventory, sales, and accounts receivable. EC files are discussed in the System User's Guide and the GCOS 6 MOD 400 Application Developer's Guide (Order No. CZ15).

A special application of EC files is their use when activating a task group. After configuration (i.e., after the CLM_USER file of configuration directives is executed) a user-written command file, START_UP.EC, attached to the root directory is executed if it is present. The START_UP.EC file might contain operator commands used to establish an applications environment for that installation.

Also, when a task group is activated whose lead task is the Command Processor, the Command Processor first executes the EC file named working_directory>START_UP.EC, if there is one. This file could contain commands used to execute the tasks of the job in sequence, without further user intervention.

EXECUTION
ENVIRONMENT

Section 4
EXECUTION ENVIRONMENT

System control of user applications and system functions is accomplished within the framework of the task group. A task group consists of a set of related tasks. The simplest case of a task is the execution of code produced by one compilation or assembly of a source program (after the code is linked and loaded).

TASK GROUPS AND TASKS

MOD 400 allows you to configure a system dedicated to interactive applications or to a combination of interactive and batch applications. This flexibility of configuration is based on the concept of the task group as the owner of the system resources it requires for execution.

By defining more than one application task group to run concurrently, you are utilizing multiprogramming. You can step through an application in sequence by causing tasks in the group to be executed one at a time; or you can multitask an application by causing tasks within the group to be executed concurrently.

Since multiple applications can be loaded in memory at the same time, contending for system resources, you must define an environment for each application so that it knows the limits of its resources. This defined environment is called a task group, whose domain includes one or more tasks, a memory pool, files, peripherals, and priority levels. By defining the total system environment to consist of more than one task group, you divide up the resources so that more than one application can run concurrently. You divide memory into memory pools, and task code of a task group is loaded only into that task group's pool and obtains dynamic memory from that pool.

By using the resources of one task group repetitively, you can run an application as a sequence of job or program steps. To do this, create a task group by a Spawn Group (SG) command to use the Command Processor, whose function is to process system-level commands. You can enter commands only in the framework of a task group whose lead task is the Command Processor. The Command Processor is activated as the lead task of the task group. One method of sequencing the steps of an application task group is to submit a command to the Command Processor to read an application command file containing a sequence of names of bound units (files of executable code), where each bound unit corresponds to a program. When a bound unit name is encountered in the file, that bound unit is loaded and executed before the next bound unit name is read. The following is an example of bound unit names in a command file:

```
        REP_DATA (The name of a program that gathers report data)
        PR_RPT (The name of a program that prints the report)
```

Another method of sequencing application steps is to issue a Spawn Task (ST) command for each task to be executed. The ST command causes the task to be loaded, executed, and then deleted. Provided the Command Processor is instructed to wait for completion of each spawned task, the tasks in the group can be executed in sequence. For example:

```
    ST 1 -EFN REP_DATA -WAIT (Spawn a task to gather report data)
    ST 1 -EFN PR_RPT -WAIT (Spawn a task to print the report)
```

This procedure can be used to attain multitasking within one task group. Consider the situation when the Command Processor is the lead task and it reads a file containing ST commands. It does not wait for the execution of the individual tasks, but continues to spawn tasks until it reads an end-of-file or &Q directive. All these spawned tasks are loaded and run concurrently in this task group, contending among themselves for the resources defined for the task group. For example:

```
        ST 1 -EFN REP_DATA (Spawn a task to gather report data)
        ST 1 -EFN PR_RPT (Spawn a task to print the report)
```

The Command Processor does not have to be the lead task of a task group. An application consisting of one task could execute in a task group whose lead task is the application task. If your application requires step control or multitasking, but you do not need the control through commands, you can generate a task group whose lead task contains Assembly language system service macro calls whose functions are analogous to the Create Group (CG), Create Task (CT), Spawn Group (SG), and Spawn Task (ST) commands.

These situations are illustrative and do not exhaust the various ways you can control program execution.

To summarize, a task group is both the owner of system resources and the context in which system control of tasking is accomplished. A task can be characterized as the execution of a sequence of instructions that has a starting point and an ending point, and performs some identifiable function. It is the unit of execution of the Executive, and its execution must be requested through the Executive software.

The source language from which task code is derived can be any of the languages supported by the Executive. Source code is compiled (or assembled) and linked to form bound units consisting of a root and zero or more overlays.

Application Design Benefits of Task Group Use

Designing an application around the task group provides intertask communication and executive control of multiple unrelated task groups.

INTERTASK COMMUNICATION

The tasks in a task group execute asynchronously in response to the interrupt-driven nature of the Executive and to a linear scan of priority levels assigned to each task group. Tasks communicate through the control structures supplied with each request for task execution.

Asynchronous tasks provide effective software response to information received from real-time external sources, such as communications or process control systems. Usually, the task that is activated to handle the interrupt from the external source has a higher priority and a shorter execution time than the task that processes the information. The task that responds to the interrupt will use the Executive to request the execution of the processing task, supplying along with the request the control structure containing a pointer to the new information to be processed. The Executive responds to the request by activating the requested task or by queueing the request if there are other requests for the execution of this task still pending.

Communications applications use a high priority task to respond to data interrupts and determine which processing task should handle the data. This high priority task uses the system to queue requests for the processing task, thereby accommodating peak-load conditions in which data is received faster than it can be processed.

In a process control system, the real-time clock might provide the interrupt that causes the high priority task to scan and update temperature, thickness, or raw material level sensors that monitor the physical status of the process. This information is passed to a processing task that determines the necessary adjustments based on the new data. A third task, having a priority between the other two, could be requested to make whatever changes are required (e.g., to change the flow rate of material entering the process by closing a valve). These two brief examples illustrate the value of priority assignments and communication facilities between tasks.

SYSTEM CONTROL OF TASK GROUPS

System control of an application based on the use of multiple task groups is important for several reasons. First, these applications can be thought of as consisting of multiple unrelated "jobs" (task groups) made up of one or more "job steps" (tasks). The sequence of task execution can be controlled by the system (Command Processor) as it processes synchronously supplied commands instead of responding only to externally supplied interrupts. The next "step" is started only when the previous step terminates.

Furthermore, if any one set of tasks does not fully use the available processing time, the system can make more efficient use of resources by rotating their use on the basis of interrupts and priority level assignments.

Finally, the use of independent task groups that are subject to system control prevents one task group from adversely affecting another. If an error occurs in one task group, it can be aborted while others continue to execute.

To summarize, system control of multiple task groups provides these advantages:

- Job and step execution sequencing
- Efficient system resource use
- Job independence.

Generating Task Groups and Tasks

The system provides tasking facilities regardless of the source code in which the application is written. Once generated, all tasks are subject to the same system controls whether written in COBOL, FORTRAN, BASIC, PASCAL, or Assembly language. Because COBOL and BASIC do not provide for tasking as part of the language syntax, the generation of tasks consisting of code written in those languages is done via commands. Although tasks written in Assembly language or FORTRAN can be generated at the control language level, these languages have a facility for generating task groups and tasks (FORTRAN) without recourse to commands. Assembly language programs use system service macro calls; FORTRAN has tasking routines.

From the overall system viewpoint, the actions of the control language in the generation of task groups and tasks are much more visible than the same capabilities in Assembly language and will be considered next.

As shown in Table 4-1, commands submitted by the operator and commands submitted by other users share some of the task group generation functions and also perform unique functions. The control commands are in three groups:

1. Commands that perform the same function whether submitted by the operator or another user (an exception being the group creation/deletion commands in the batch mode)

2. Commands entered only by the operator

3. Commands contained within the content of an existing task group request.

Characteristics of Task Groups and Tasks

Task groups and individual tasks can be originated in either of two ways: they can be created or spawned. The choice depends on application design considerations as well as the intended functions.

There are important differences between tasks (and task groups) that are generated by a create function and those originated by a spawn function. Created task groups and tasks are permanent; they remain available in memory until they are explicitly removed. Spawned task groups and tasks are transitory; they perform a function and disappear.

Created task groups and tasks are passive; they must be explicitly requested to execute in order to perform their intended function. Spawned task groups and tasks cannot be requested. The spawning of a task group or task is equivalent to a create-request-delete sequence of control language commands: the task group or task is defined, provided with system resources and control structures, executes, terminates, and has its resources deallocated, all in one continuous process.

FORTRAN or Assembly task code may cause extensive action in its own behalf, as when application task code requests a system service or the execution of another task while awaiting the completion of the requested task. Each task that requests another supplies the address of a control structure through which the issuing task and the requested task can communicate, and which the Executive software uses to coordinate task processing.

Table 4-1. Task Group and Task Functions Possible From Interactive or Batch Modes[a]

	User Commands[b]		Operator Commands[b]	
Function	Interactive	Batch	Interactive	Batch
Create Group	Yes	No	Yes	Yes
Enter Group Request	Yes	Yes	Yes	Yes
Delete Group	Yes	No	Yes	Yes
Abort Group	Yes	No	Yes	Yes
Spawn Group	Yes	No	Yes	Yes
Bye	Yes	Yes	No	NA
Enter Batch Request	Yes	Yes	NA	Yes
Suspend Group	Only operator commands exist for these functions		Yes	NA
Activate Group			Yes	NA
Abort Group Request			Yes	NA
Suspend Batch Group			Yes	
Activate Batch Group			NA	Yes
Create Batch Group			NA	Yes
Delete Batch Group			NA	Yes
Abort Batch Request			NA	Yes
Abort Batch Group			NA	Yes
Create Task	Yes	Yes	No operator commands exist for these functions	
Delete Task	Yes	Yes		
Enter Task Request	Yes	Yes		
Spawn Task	Yes	Yes		

[a] The Command Processor executes in both interactive and/or batch modes.

[b] NA means Not Applicable.

Task Group Identification

Each task group has a unique identifier. Vendor-supplied system task group identifiers begin with a $ as shown below:

Task Group ID	Function
$B	Batch
$D	Debug
$L	Listener
$P	Deferred Print/Punch
$S	System

NOTE

The Multi-User Debugger does not require the dedicated system task group $D.

The identifier for a user task group in the Create Group or Spawn Group command is a two-character name that should not have the $ as its first character. The identifier (or group-id) can be indicated or implied in commands to designate what task group is to be acted upon. The operator can include the task group identifier in responding to messages from the task group.

MEMORY USAGE

At system startup the Configuration Load Manager (CLM) reads a file of directives, sets up memory pools from the specifications supplied, and indicates to the Loader what system and user-written software is to be resident for the life of the system. The file of CLM directives can be created by the system builder with the Editor alone or with the help of the interactive CLM directive generation program, M4_SYSDEF.

M4_SYSDEF automatically calculates the individual user memory pools based on information supplied by the system builder during the interactive session. After the system has been in operation, it may be desirable to reconfigure the memory so that it meets your requirements more precisely. This can be done by hand-tailoring the file of CLM directives. Refer to the System Building and Administration manual for details on the use of the M4_SYSDEF utility.

The number of memory pools to be used and their characteristics are specified by MEMPOOL and SWAPPOOL CLM directives.

Memory Management and Protection

The system (hardware and software) provides a memory management and protection facility. The memory management and protection facility performs the following functions:

- Dynamically allocates memory to guarantee each task group (user) its own address space

- Optionally protects multiple users from each other and the system from the users.

MOD 400 offers two forms of memory management, depending on the memory pool configured. Swap pools provide a segmented memory management environment in which all users are protected from each other and in which memory requirements of individual users do not have to be predetermined. Online memory pools provide a minimum overhead fixed-partition environment suitable for real-time applications with well defined memory requirements.

SWAP POOLS

The Swap Pool Memory Manager uses the capabilities of the Memory Management Unit (MMU) to assign objects to segments. It may cause segments to be swapped out to a swap file on disk in order to make physical memory available to competing users. Swap pool memory management can move segments to real physical memory in order to compact the utilization of physical memory and eliminate fragmentation.

Segments

Swap pool memory management is based on the concept of a segmented address space that is mapped onto real memory by the firmware and hardware of the MMU. The unit of memory allocation is a segment. A segment is a variably sized area of memory that usually consists of a logical entity such as a procedure. The memory management facility treats all addresses generated by the central processor as segment-relative addresses; it maps them through the MMU array to absolute physical addresses.

The MMU supports up to 31 segments, 16 of which can be up to 4K words (K = 1024) in size ("small" segments) and 15 of which can be up to 64K words in size ("large" segments). Segments have the following characteristics:

- The 16 small segments are numbered from 0.0 to 0.F; the 15 large segments are numbered from 1 through F.

- No segment can be less than 256 words long; segment size increases in increments of 256 words to a maximum of 4K for small segments or 64K for large segments.

- Although users can assign any of the large segments (1 through F) to a bound unit when it is linked, the availablity of a segment depends on the system configuration. All small segments and often some large segments are reserved for system use; the actual number reserved is established at system generation. The identity of segments available to the user should be obtained from the system administrator.

- Each segment is described by a two-word segment descriptor that contains the segment's starting physical address, its length (in units of 256 words), and its access rights for each ring (refer to "Segment Ring Protection" later in this section).

Segment/Bound Unit Relationship

User programs are linked to form bound units that are loaded into memory. At link time, the user can specify the segment(s) to which the bound unit is assigned. The user's physical address space is not necessarily contiguous; memory requirements are satisfied on a segment basis rather than on a user basis.

Swappable Segments

The memory management facility acquires space for a given segment from whatever portion of free memory is available.

If not enough space is available to obtain memory for the needed segment, the system attempts to obtain memory by swapping out lower priority tasks that are wating on an event. If this action still does not produce enough memory, the requesting task is swapped out until sufficient space becomes available.

A task is swapped out under one of the following conditions:

- It is waiting on an event that is of potentially long duration and swap pool memory is required by any competing task

- Memory is required to roll in a higher priority task

- It has been suspended by the operator.

A task is swapped back in when swap pool memory is available either immediately or by swapping out tasks waiting on long duration events, or by forcing lower priority tasks to be swapped out. A task is swapped back in when any event on which it was waiting has completed or when it is reactivated by an operator.

Sharing Segments

If a bound unit is linked with the SHARE option, the root segment of the bound unit is available to any user who has the proper access to the bound unit file. The root segment should contain reentrant code, and the the bound unit should have no fixed overlays. Floatable overlays of the sharable bound unit are shared only when an Overlay Area Table (OAT) is used. (Fixed and floatable overlays and the OAT are described later in this section.)

To permit intertask communication, certain group global segments are shared by all the tasks in a task group. The group work space and the group system space are both shared by all the tasks in a task group.

Additionally, task "private" segments are shared if the task forks. (A task forks if it issues a Create Task or Spawn Task command with an entry point rather than a pathname definition.) The forked tasks share the same segments. The forked tasks have the same visibility of and copy of forked segments until either task modifies its address space.

SEGMENT RING PROTECTION

Access to memory segments is controlled through a hardware-supported protection mechanism that assigns each executing task to a ring of privilege. During the linking of a bound unit, the user can assign access attributes to each bound unit that indicate whether a task executing in a particular ring of privilege can read, write, and/or execute code in the code or data segment of the bound unit.

System tasks execute in ring 0 (privileged state). User tasks can execute in rings 1 (privileged state), 2, and 3 (unprivileged state). The ring in which a user task executes is not a function of the linking of the bound unit; all user tasks assigned to the swap pool are created in ring 3 automatically. User tasks assigned to other online memory pools may be created to run in ring 1 or 2 (default), depending on the configuration of the memory pool.

Every attempted access to a segment is checked for access in the executing task's ring of privilege. The system compares the ring number of the executing task with the access attributes of the segment to be accessed. An access violation trap occurs if a user application attempts to access one of its own segments or a segment outside its own address space, without having the proper segment access rights.

SEGMENTED BOUND UNITS

Task code is derived from programs written in some source language that are compiled or assembled to form object units (also called compilation units). One or more object units are linked to form a bound unit.

A bound unit is composed of one or two segments. The code segment is composed of one or more load elements. A load element is composed of one or more compilation units. The initial load element is called the root. The root must be resident when the bound unit is being executed. A load element that replaces another load element when loaded into memory is called an overlay.

You can direct the Linker to perform the following actions on bound units:

- Map the code and data into the same segment or into separate segments
- Optionally, associate a specific segment number or numbers with a bound unit. If you do this, you must be careful to avoid segment conflicts in the configuration and application environment in which the bound unit is to run.
- Specify ring access rights.

You have a maximum of 11 large segments available when you construct a task's address space. Frequently, you have fewer large segements available, depending on the configured size of system global space.

Segmented Bound Unit Overlays

To minimize the amount of memory required to execute a bound unit containing application code, you can cause the bound unit to be created as a series of overlays by specifying overlay directives to the Linker. Each bound unit consists of a root and, optionally, one or more related overlays. The maximum number of overlays is 1024 (numbered 0 through 1023).

You can also use macro calls to create an OAT for a bound unit. The overlays specified in the OAT can be shared among all tasks sharing the same bound unit. An OAT created for a task's initial bound unit can be used by any attached bound units.

Segmented Reentrant Bound Units

The Linker produces bound units that can optionally be reentrant. In a reentrant bound unit, the code and data are in separate segments. At link time, you can specify that the bound unit is to have separate code and data segments; you can optionally specify the access attributes (read, write, and execute) of each segment. The default access attributes are ring 3 read and execute access for both code and data, ring 0 write access for the code segment, and ring 3 write access for the data segment.

Sharable Segmented Bound Units

Sharable bound units make reentrant code available to multiple tasks executing in the same memory pool. The user indicates that a bound unit is sharable at link time. If a bound unit is sharable, the descriptor for the root segment of the bound unit is placed in a portion of memory where it is accessible to all tasks in all groups.

Each task sharing the bound unit refers to a common copy of the root segment. An example of a sharable bound unit is the line editor. Multiple users share code in the root segment of the line editor bound unit; each user has his own copy of the data.

Sharable bound units are transient and are loaded into the executing task's memory pool during processing. A counter is incremented each time a request is made for the bound unit and is decremented when the request has been satisfied. Once the counter is decremented to zero, the space occupied by the bound unit is released.

A bound unit can also be linked as globally sharable. Globally sharable bound units are loaded in the system pool and can be accessed by any task in any group. However, system pool memory is a critical resource, and the use of globally sharable bound units requires careful planning and control to prevent exhaustion of that resource.

TASK ADDRESS SPACE

The task address space is used to define a task's boundaries; i.e., its visibility within the collection of executing tasks. The following elements constitute a task's address space:

- Bound unit
- User stack segment
- Dynamically created segments
- Group work space
- Group system space
- System global space.

Bound Unit

During its execution life, a task executes one or more bound units. The initial bound unit to be executed is the one specified when the task is created or spawned. Other bound units (if any) are attached or loaded through the Bound Unit Attached ($BUAT) and Bound Unit Load ($BULD) macro calls. All segments included in the bound unit(s) executed by the task are within the task's address space and are accessible to the task (consistent with the ring protection specifications).

User Stack Segment

A user stack segment is provided for each task associated with a bound unit having the stack option. The user stack area is available to users as a work area through the hardware stack instructions.

Dynamically Created Segments

During execution, a task can extend its address space by creating segments using the Create Segment ($CRSEG) macro call. These dynamically created segements become part of the issuing task's address space.

Group Work Space

Included in the task's address space is up to two large segments known as the Group Work Space (GWS). The GWS is common to all tasks in a given task group. Tasks obtain blocks of memory from the GWS in units of 32 words when they issue Get Memory ($GMEM) macro calls. All tasks in the taks group have read, write, and execute access to the GWS. The GWS grows dynamically, as requests for memory are issued, up to a maximum size of 128K words if the adjacent segment descriptor has not been allocated for some other segment or 64K words if the adjacent segment descriptor has been allocated.

Group System Space

Also included in a task's address space is the Group System Space (GSS). One GSS is provided for each task group. The system control structures used to support a task group and its member tasks (e.g., file control blocks, bound unit descriptors for nonsharable bound units, LFTs, and LRTs) are allocated from the GSS. Access to the GSS is read and execute access from ring 3 and write access from ring 0. The GSS grows dynamically, as requests for memory are issued, up to a maximum size of 128K words if the adjacent segment descriptor has not been allocated for some other segment or 64K words if the adjacent segment descriptor has been allocated.

System Global Space

System Global Space (SGS) consists of the fixed system area (permanently configured memory) and the system memory pool. A task's address space includes the segments required for system global space. System code and data are distributed in the task address space. Note, however, that a user task can gain write access to system space only through the system service macro call interface. (A user task must be in ring 0 before it can modify system data.)

System Representation of Task Address Space

Figure 4-1 is an example of the mechanism used by the system to represent a task's address space. The following points should be noted:

1. The layout of memory is logical, not physical.

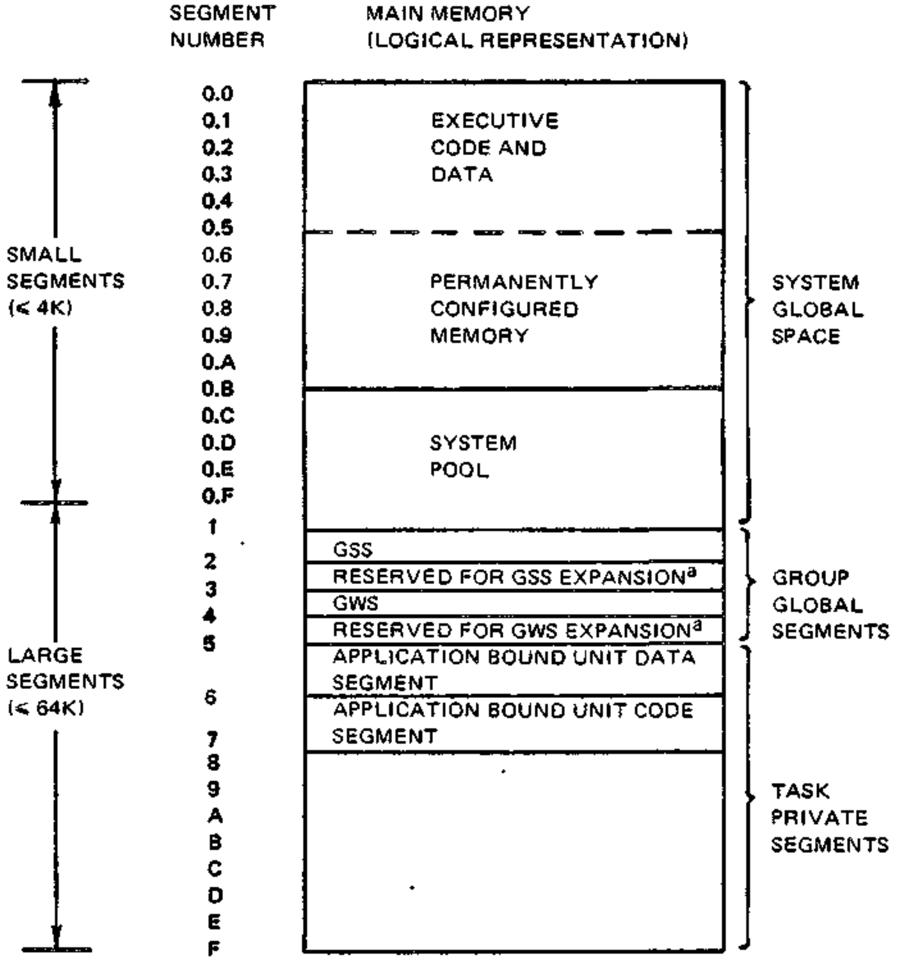
2. The layout applies only to this example; it is possible to generate system whose layout is different from that shown in Figure 4-1.

3. The segments available to the user for his/her bound units are 6 through F. If the GSS segment requirements are less than or equal to 64K, segment 3 can be used. If the GWS segment requirement is less than than or equal to 64K, segment 5 may be used.

4. One copy of segments 0.0 through 1 exists in the system in this example. These segments contain the SGS. All tasks in the system can access these segments. To modify these segments, a task must be in ring 0.

5. Segments 6 through F are unique to the task, unless they are being shared (because they contain a sharable bound unit or the task has been forked). If one of these segments is being shared, each task sharing the segment accesses the same copy of the segment. When a segment number is assigned by the Memory Manager (e.g., for a bound unit that did not have a segment assignment specified when it was linked), then the lowest available segment (or segments for objects of size greater than 64K) begining with the GWS segment plus 2 (segment 6 in this example) will be used. If all of segments from GWS+2 through large segment F have been used, then the segments GWS+1 (segment 5 in this example) and GSS+1 (segment 3 in this example) are allocated in that order.

6. Only one copy of the GWS segment (segment 4 in this example) exists per task group. All tasks in the task group have unlimited access to this segment. only one copy of the segment that contains the GSS (segment 2 in this example) exists per task group. All tasks in the task group have read and execute access to this segment. Both the GWS and GSS segments are dynamically expanded as demands are made on them. Each can grow to a maximum of 128K if the adjacent ascending segment descriptor (segment 3 for GSS and segment 5 for GWS in this example) has not been previously allocated to contain a task private segment.



[a]CAN BE USED BY USER TASKS IF GSS/GWS NEVER EXCEEDS 64K.

Figure 4-1. Task Address Space

Allocating and Deallocating Segments and Bound Units

Each task is associated with at least one bound unit. The initial bound unit with which a task is associated is specified at the time the task is created or spawned. At this time, the segment is created/allocated in memory, and the root is loaded in this segment.

If the bound unit was designated as sharable at link time and is currently residing in memory, no loading takes place. The requesting task shares the bound unit already in memory and the bound unit user count is increased by one. If the bound unit is not in memory, it is loaded.

Execution of a task begins with the specified bound unit. During the execution of this bound unit, the user uses system service macro calls to load or attach another bound unit. Loading or attaching a bound unit causes the allocation and loading of the segment containing the root of the requested bound unit. (The difference between loading and attaching is that loading returns the entry point of the root segment to the issuing task, while attaching starts the execution of the bound unit root segment at the entry point.) Up to eight bound units can be attached. The availability of segment descriptors may limit the user to fewer than eight attached bound units.

During its execution, a task can issue a system service macro call to request the creation of a segment to be associated with the task's initial bound unit or any other attached/loaded bound unit. The macro call can either specify a segment number or allow the system to select the number in accordance with the specified size.

ALLOCATING SEGMENTS AND BOUND UNITS

The allocation of memory for a bound unit depends on whether the bound unit is nonsharable or sharable.

For a nonsharable bound unit, each logical segment is uniquely mapped to a physical segment in memory. Unless the task is replicated or the segment is in an OAT, two or more tasks wishing to concurrently use a nonsharable bound unit each receive a copy of the bound unit. If a segment assignment was not made when the nonsharable bound unit was linked, the bound unit is assigned the next available segment descriptor(s) at GWS+2 or above in the issuing task's address space. Thus, each copy of the bound unit may have a different segment assignment.

If more than one task is executing a sharable bound unit, only one copy of the segment containing the root is allocated in memory. All tasks use this single copy. Overlays of the bound unit can be shared if an OAT is used. If the root was separated into a code segment and a data segment, only the code segment is considered to be the root. Except for forked tasks, each user has a separate copy of the data segment.

If a segment assignment was not made when the sharable bound unit was linked, it is assigned a segment number or numbers based on the segment descriptors available in the address space of the task that initially loads the bound unit. The bound unit must be accessed under the same segment numbers by any concurrent users of that bound unit. If the segment utilization of the second and subsequent tasks that attempt to load the sharable bound unit conflicts with its segment number or assignment, then an error is return when the tasks attempt to load the bound unit; the tasks are not given addressability to the sharable bound unit.

DEALLOCATING SEGMENTS AND BOUND UNITS

You can explicitly deallocate a user-created segment by issuing a Delete Segement ($DLSEG) macro call. You can deallocate bound unit segments by detaching any but the initially assigned bound unit. A segment can implicitly be deallocated from real memory as the result of the task being deleted or swapped out. It is reallocated when the task is swapped back in.

All tasks that are assigned to the swap pool are eligible to be swapped out. The swapper swaps out one or more tasks until sufficient memory is freed to satisfy a request. When the task is swapped back in, its segments are allocated wherever there is free memory.

Online Pools

Online pools become part of the system resources owned by task groups. There are two types of online pools: exclusive and nonexclusive.

EXCLUSIVE ONLINE POOLS

An exclusive pool is one whose boundaries do not overlap those of other pools. The lower part of Figure 4-2 shows a configuration of five online exclusive pools. Each pool is to be used for the task of one or more task groups. The pools are shown empty as they would be at the end of the configuration process.

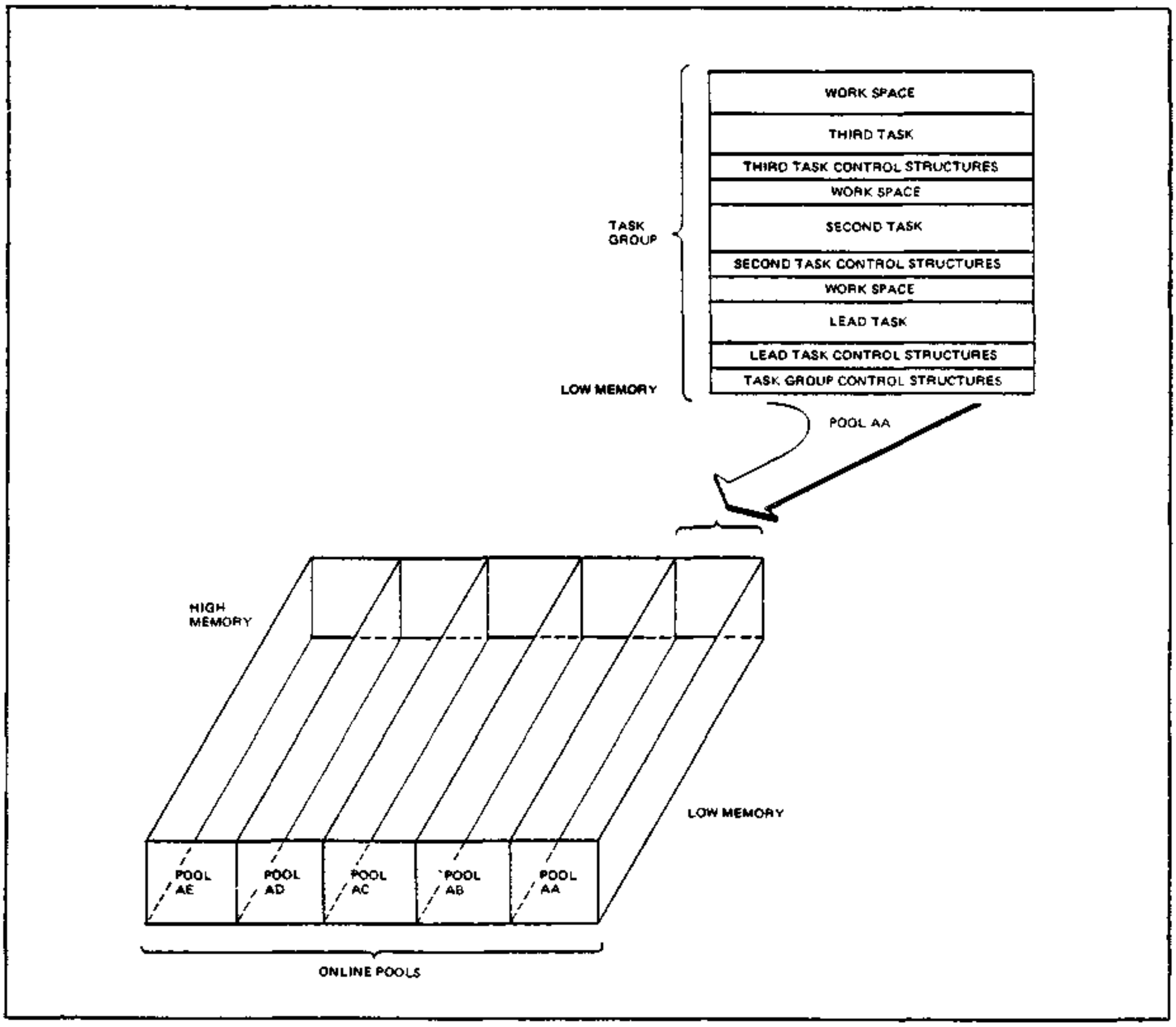


Figure 4-2. Exclusive Memory Pools and Contents

The upper part of the figure is an idealized picture of the contents of Pool AA at some instant during processing and shows the memory layout for three concurrent tasks. The figure does not accurately reflect the fact that memory is allocated and returned (in Assembly language programs) dynamically when needed, and work space might not be contiguous to the task code that requests it. Also, memory is allocated in contiguous blocks according to an algorithm that uses multiples of 32-word blocks to calculate the amount of space to assign to a pool. The "holes" that would normally be present as a result of the operation of the algorithm are missing. The Memory Manager adds one plus $AF words for control information to the requested amount of space, divides this value by 32, rounds up to the next whole number, and allocates this calculated amount of memory to a task group.

NONEXCLUSIVE ONLINE POOLS

A nonexclusive pool set is a set of pools whose boundaries overlap those of other nonexclusive pool sets so that some memory locations are common to both pool sets. Figure 4-2 shows two pool sets that are alternative definitions of the same physical memory area.

SHARING MEMORY POOLS

There are two ways of sharing memory pools. The first method involves assigning two or more task groups to the same pool. As these tasks execute, they contend for the same memory space. They should, therefore, be designed so that they can be suspended or take some alternative action when no additional memory is available.

The second method of allowing task groups to share memory involves the definition of nonexclusive pools sets. Figure 4-3 shows how this might be done.

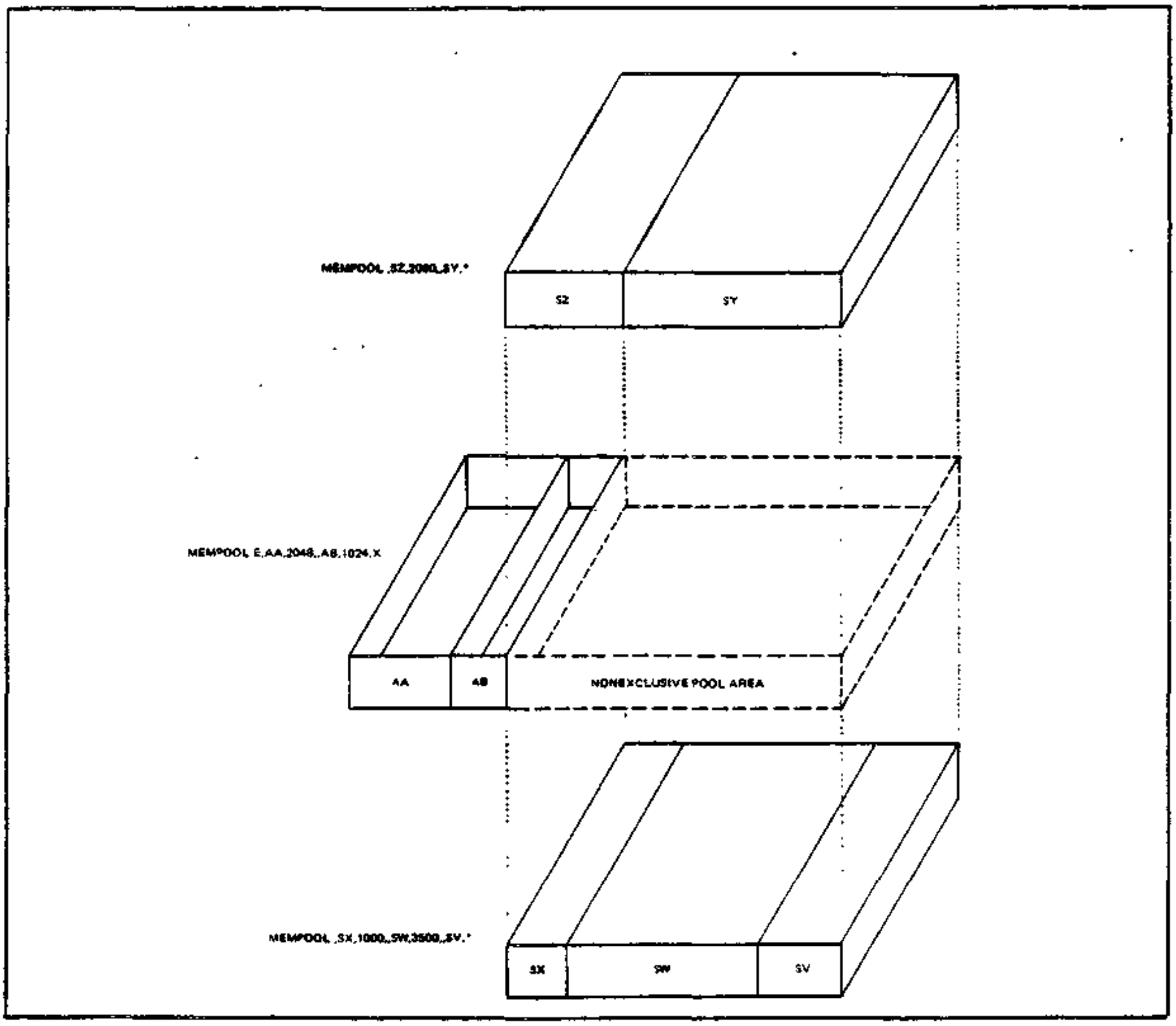


Figure 4-3. Exclusive and Nonexclusive Pool Sets

Pool sets SX/SW/SV and SZ/SY represent alternate definitions of the same physical area of memory. This method of memory use has the advantage of providing flexible and efficient use of resources at any one time. But it has the disadvantage that, unless task group requests are very carefully planned, software deadlock (memory usage conflicts) can occur--the Executive does not prevent it. If task groups using the SZ/SY pool set never execute until those using the SX/SW/SV pool set have terminated, there will never be a problem of deadlock, if only one task group uses each pool.

However, assume that a task group assigned to pool SY is generated and acquires some of the pool space. Then, a task group assigned to pool SW is generated and acquires some space. If each group requested its remaining space and was willing to wait until the space was available, a deadlock would occur--neither task would ever complete.

The surest way to avoid potential memory usage conflicts is to define all online pools as exclusive pools, and additionally to confine pool use to one task group.

If serial usage for a pool is specified, the Executive will not allow a group that uses the pool to be created when another group is already using the pool.

Fixed System Area

The following software components and data structures will be located in the fixed system area after the configuration process is complete:

- Basic Executive plus resident overlays (RESOLA directive)
- User-written or vendor-supplied extensions to the Executive (LDBU or DRIVER directive)
- Device drivers
- Intermediate request blocks needed for task groups (SYS directive)
- Trap save areas (SYS directive)
- Overlay area(s) for system software (SYS directive)
- File control structures (File Description Block (FDB) for nondisk devices).

The Executive area is fixed--its structure remains the same for the life of the system--in contrast to other memory areas whose usages can vary. Almost all code loaded into this area is reentrant so that a single copy of the code is available to multiple users, thus minimizing memory requirements.

System Pool Area

The area adjacent to the resident software area is called the system pool. This area contains the system task group. In addition, the system pool accommodates the following elements:

- Current function invoked by an operator command
- Extended Trap Save Areas (TSAs) needed during processing
- Control blocks for all tasks and task groups (TCBs and GCBs)
- Globally sharable bound units
- File System directory and file definition blocks
- Public buffer pools
- Memory control blocks for swap pool segments.

SYSTEM TASK GROUP

The system task group differs from other task groups in the following ways:

- Cannot be aborted or suspended
- Always has read and write access to all of memory
- Handles all system dialog (including operator commands) through the designated operator terminal
- Never terminates, so it cannot be requested.

FILE CONTROL STRUCTURES IN THE SYSTEM POOL AREA

The elements in the system pool area that are used for file control consist of:

- File Description Block (FDB)
- Buffers for sharable files.

Pool Attributes

You can exercise more control over memory usage by providing online memory pools with specialized attributes, as described in the following paragraphs.

PROTECTED MEMORY POOLS

A memory pool can be "protected" if it is so specified at configuration (by a CLM directive). A protected pool is one into which a task running in another pool cannot write. Through the use of the MMU, the Executive prevents write intrusion by foreign tasks. Such a task receives an error notice from the Executive when an intrusion is attempted.

The special size constraints that apply to protected pools are described in the System Building and Administration manual.

Protected pools require the MMU.

CONTAINED MEMORY POOLS

Any memory pool but the system memory pool can be "contained" if it is so specified at configuration. Tasks running in a contained pool are prevented from writing outside their own pool area. The constraints that apply to the size of contained memory pools are the same as those that apply to protected memory pools.

Contained pools require the MMU.

UNPRIVILEGED MEMORY POOLS

At configuration, any memory pool except the system pool, swap pool, or batch pool can be declared "privileged." A task running in an unprivileged pool cannot execute privileged instructions and will trap if such an execution is attempted. A memory pool can be declared unprivileged regardless of whether or not the configuration has an MMU (i.e., the privileged function is independent of the memory protection function).

The following Assembly language instructions are privileged:

ASD	IO	LEV	WDTF
CNFG	IOH	RTCF	WDTN
HLT	IOLD	RTCN	

The system pool is always privileged and the attribute cannot be altered.

The swap and batch pools are always unprivileged and the attribute cannot be altered.

Exclusive and nonexclusive pools are unprivileged unless specified to be privileged.

SERIAL-USAGE MEMORY POOLS

An exclusive or nonexclusive memory pool can be declared serial-usage. If so declared, such a pool can be used by only one task group at a time.

MULTIPOOL MEMORY PROTECTION

On a system having an MMU, the system builder can specify the write protection and/or containment that the system is to provide. At CLM time, the system builder selects one of the following options; the option selected prevails until the system is reconfigured.

1. No protection or containment (i.e., no utilization of the MMU).

2. Protection of the system memory pool and/or containment of the batch memory pool.

3. Protection and/or containment of selected memory pools in addition to those of option 2.

Protection applies to memory pools and not to task groups. Groups sharing a memory pool are only protected from each other in swap pools. Also any group in a memory pool is not secure from intrusion if the pool is a nonexclusive pool.

If a task group is to be protected from all other groups, it must be the only group using an exclusive memory pool, and the system builder must specify option 3 software protection at system configuration.

MEMORY LAYOUT

To obtain efficient use of memory and of the MMU, the CLM sorts the memory pools in a configuration as follows:

1. The system pool is in the first available memory after the system data structures.

2. Any online pools are configured next. Within the online pools:

    a. If a swap pool is configured, it follows the system pool.

    b. Nonprotected, noncontained pools are next, in order of increasing size.

    c. Protected and/or contained pools come last in order of size; the smallest one comes first.

    d. For purposes of memory allocation, all nonexclusive pools are considered collectively to be a single pool.

    e. Independent pools are configured last.

SELECTING MEMORY POOL ATTRIBUTES FOR TASK GROUP EXECUTION

The different types of memory pools provide you with the means to respond to the unique demands of multiple application programs. Through the use of memory pools, you can at once exercise control over memory usage and at the same time provide individual task groups with specialized protection attributes.

The degree to which the system can efficiently and effectively handle the concurrent execution of multiple task groups depends on the number and type of memory pools available for use.

Cases 1 and 2 examine the considerations involved in the selection and use of memory pools:

Case 1:

Your program consists of a real-time data application program. The program must coexist with other user applications.

The data application program accepts data based on unpredictable external stimuli. The application permits a "saturation" effect to occur when data collection exceeds the effective rate of processing. The occurrence of saturation represents a signal to the application to initiate a data selection strategy.

You should select an online pool of sufficient memory size to control the maximum desired amount of data allowed to accumulate.

Case 2:

Multiple applications with strict integrity requirements are to coexist with programs under development test.

You can choose to load each of the applications with integrity requirements into a memory pool that has been designated as protected. This will ensure that the programs under test do not accidentally modify data in the programs to be protected.

BOUND UNITS

Task code is derived from the source language of programs that are compiled or assembled to form object units. One or more object units are linked to form a bound unit that is placed on a file. The bound unit is an executable program that can be loaded into memory. A task represents the execution of a bound unit. Each bound unit consists of a root segment and any related overlay segments.

Sharable Bound Units

Using sharable bound units is a way of minimizing application task group memory requirements while making reentrant code available to multiple tasks. Unlike permanently resident bound units that are loaded during system configuration, sharable bound units are transient in memory and are loaded during processing. A counter is incremented each time a request is made for the bound unit, and the unit remains in memory as long as a task is using the code. As soon as the counter is decremented to zero, the system pool space occupied by the bound unit is returned to available status.

Operator commands can be used to load and then unload a globally sharable bound unit.

To be recognized as sharable by the Loader and loaded into the system memory pool, the bound unit must have been so marked by the Linker in response to a GSHARE directive when the bound unit was linked. To be recognized as sharable by the Loader and loaded into a user memory pool, the bound unit must have been so marked by the Linker SHARE directive at link time. As the system pool memory is a critical resource, it is recommended that you use the SHARE directive to indicate a sharable bound unit.

Sharable bound units and the Executive extensions that are loaded when the system is configured using the LDBU CLM directive differ in one major respect. Executive extensions can be accessed symbolically by any task, but a sharable bound unit must be accessed as a bound unit. When an Executive extension is loaded during system configuration and it is made permanently resident, its symbols are included in the system symbol table. Since a sharable bound unit is loaded after the system has been configured and is transient, no entry for it is made in the system symbol table and it must be accessed as a bound unit.

Table 4-2 compares permanently resident Executive extensions and transient sharable bound units.

Overlays

To minimize the amount of memory required to execute a bound unit containing application code, you can create the bound unit as a root and one or more overlays at link time. The use of overlays requires careful planning so that required code is not lost or repetitively loaded.

Overlays can be loaded at a fixed displacement from the base of the root (nonfloatable overlay) or into a block of memory allocated explicitly by you or implicitly by the system (floatable overlay).

Table 4-2. Comparison of Executive Extensions and Sharable Bound Units

Characteristics	Executive Extension	Sharable Bound Units
Multiple Users	Yes	Yes
Permanent Resident (Fixed area)	Yes	No
Temporary Resident (Dynamic area)	No	Yes
Symbols in System Table	Yes	No
Accessed Symbolically?	Yes	No
Can Have Overlays?	No	Yes
Called by Bound Unit Name	No	Yes

NOTES

If the extension is an Assembly language bound unit, it may have within it sections of code or control structures controlled by semaphores that would be accessible to other Assembly language tasks.

Overlays are not sharable unless Overlay Area Tables (OATs) are used.

The Executive does not "remember" extensions by their names; a request for one by name results in another copy being brought into memory.

You can create a set of overlay areas and have the system load floatable overlays into them. The system will manage the availability of free areas and locate available copies of the requested overlays.

NONFLOATABLE OVERLAYS

A nonfloatable overlay is loaded into the same memory location relative to the root each time it is requested. Object units whose code is to be loaded as nonfloatable overlays must be defined as fixed overlays by the Linker OVLY directive. When the root of a bound unit having fixed overlays is loaded, the Loader allocates a container (segment or memory block) large enough to hold the root and any of its fixed overlays.

Assembly language programs can use system service macro calls to load and execute nonfloatable overlays. COBOL and BASIC programs can use CALL/CANCEL statements to control nonfloatable overlays. FORTRAN programs must link a user-written Assembly language Overlay Manager with the application program.

FLOATABLE OVERLAYS

A floatable overlay is linked at relative Location 0 and can be loaded into any available memory location. Floatable overlays must meet the following criteria:

1. The overlay must not contain external definitions referenced by the root or another overlay.

2. The overlay can have Immediate Memory Addressing (IMA) references either to itself or to the root and/or any nonfloatable overlay. The two options are mutually exclusive.

3. The overlay must not make displacement references to the root or any other overlay.

4. The overlay must not contain external displacement references that are not resolved by the Linker.

The application program can use one or more areas of available memory for the placement of floatable overlays. The program can deal with memory management in the following ways:

1. Allow the system to place the overlay in an available memory block (allocated from the users fixed partition pool or from this GWS segment if loaded in a swap pool).

2. Create a set of overlay areas (using system service macro calls) and allow the system to manage the areas and locate the requested overlays. Such overlay areas are created in a memory block from the same fixed-partition memory pool in which the root segment was loaded or, if loaded in a swap pool, the segment(s) allocated from the root is expanded to contain the created overlay area.

3. Perform its own memory management by linking a user-written Assembly language Overlay Manager with the root of the bound unit. In this instance, the user may choose to have a floatable overlay occupy a part of or an entire memory block (or segment if in a swap pool).

LINKER ASSOCIATED OVERLAYS

Floatable and nonfloatable overlays are defined through the Linker. When using the Linker, forward references can be made to symbols defined in object units to be linked later. Backward references can be made to symbols previously defined, provided that the defined symbols were not purged from the Linker symbol table by a Linker Base or Purge directive. Since the specification of the Base directive removes from the Linker symbol table all previously defined and unprotected symbols that are at locations equal to or greater than the location designated in the Base directive, you must either define all symbols that you want to preserve in a nonoverlaid part of the root or protect these symbols by using the Linker Protect directive.

Floatable overlays can refer to fixed addresses in the root or nonfloatable overlay, but cannot refer to addresses in another floatable overlay.

When a root or an overlay of a bound unit is loaded, the Loader examines the attribute tables associated with the bound unit if an alternate entry point is specified. The Loader tries to resolve any references to symbols that remain unresolved by searching the system symbol table (i.e., the resident bound unit attribute table); it cannot resolve any references to symbols that do not exist in that table (Linker symbol tables do not exist at load time).

Figure 4-4 shows the relative location in memory of Memory Pool AA. Figure 4-5 is the layout of overlays in Memory Pool AA. The Linker directives to create and specify the location of these overlays are described in the Application Developer's Guide.

Overlay areas are fixed size areas of memory; their use is controlled through an Overlay Area Table (OAT). Overlay areas (and OATs) are part of the bound unit. If the bound unit is sharable, the overlays can be shared with other tasks in the task group or with other tasks in other task groups. Overlays can also be shared if the bound unit is replicated through the -SHARE argument of the Create Task command.

You create an OAT through a $CROAT system service macro call. You reserve an overlay area and execute the overlay by means of a $OVRSV macro call. You exit from the overlay through a $OVRSL macro call.

As an example of overlay area use, assume that you desire to share both the root and overlays of a sharable unit whose structure is shown in Figure 4-5.

When the root is loaded, the largest contiguous amount of memory necessary to accommodate the root and all nonfloatable overlays is allocated. Except for space for any floatable overlays, no other memory requests need be made. In Figure 4-5, this memory area begins at relative Location 0 of the root and continues to the end of object unit OBJD. The root consists of object units OBJ1 and OBJ2. When loaded, OBJ5 of overlay ABLE will replace the previously loaded OBJ2 code of the root. Similarly, the overlay locations were specified so that OBJC of overlay ZEBRA will replace part of OBJB.

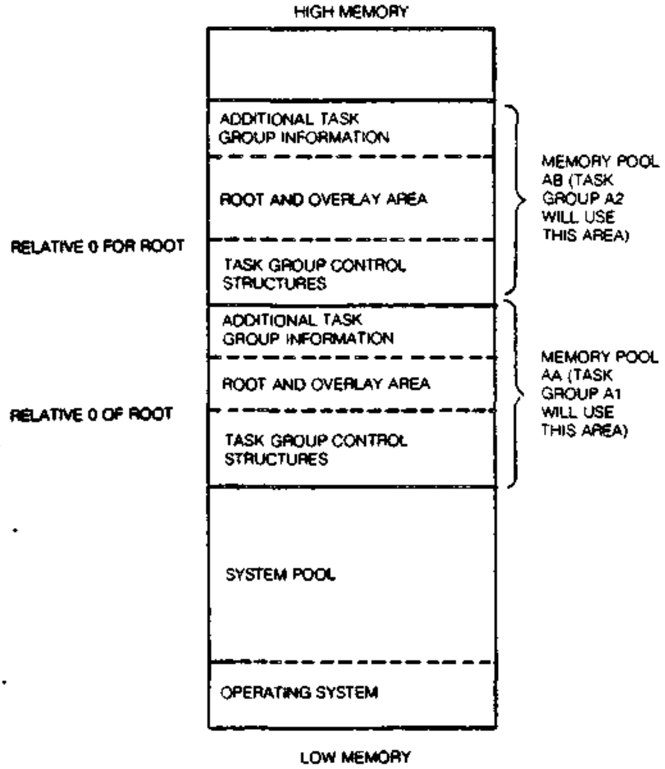

HIGH MEMORY
ADDITIONAL TASK GROUP INFORMATION
ROOT AND OVERLAY AREA
TASK GROUP CONTROL STRUCTURES
MEMORY POOL AB (TASK GROUP A2 WILL USE THIS AREA)
RELATIVE 0 FOR ROOT
ADDITIONAL TASK GROUP INFORMATION
ROOT AND OVERLAY AREA
TASK GROUP CONTROL STRUCTURES
MEMORY POOL AA (TASK GROUP A1 WILL USE THIS AREA)
RELATIVE 0 OF ROOT
SYSTEM POOL
OPERATING SYSTEM
LOW MEMORY

Figure 4-4. Relative Location in Memory of Memory Pool AA

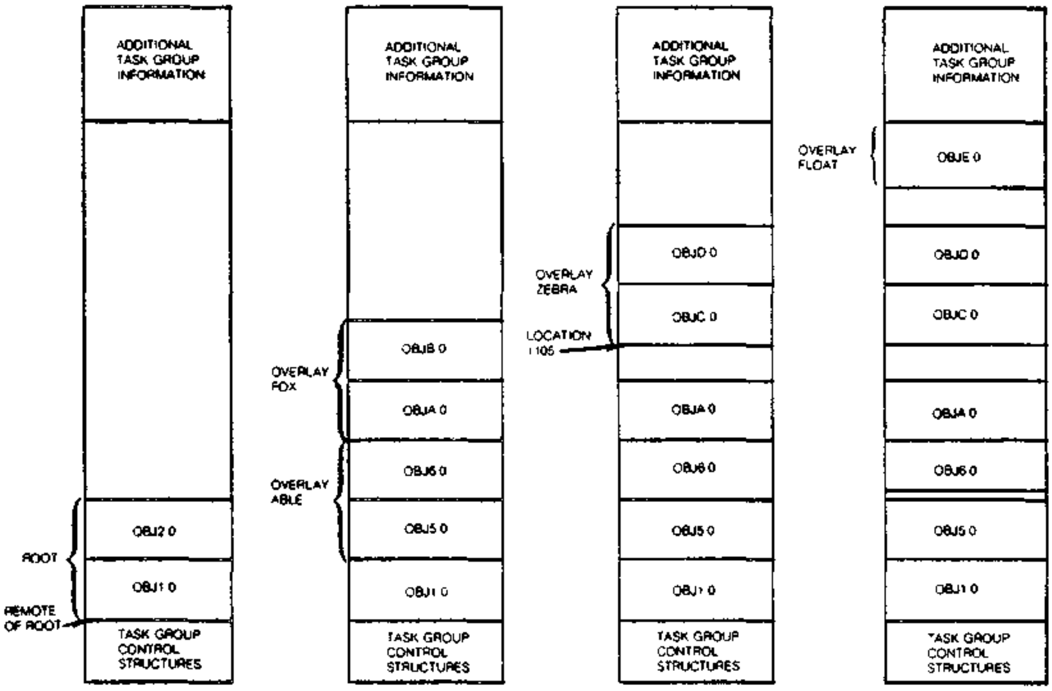

ADDITIONAL TASK GROUP INFORMATION
OBJ2 0
OBJ1 0
ROOT
REMOTE OF ROOT
TASK GROUP CONTROL STRUCTURES
ADDITIONAL TASK GROUP INFORMATION
OBJB 0
OBJA 0
OVERLAY FOX
OBJ6 0
OBJ5 0
OVERLAY ABLE
OBJ1 0
TASK GROUP CONTROL STRUCTURES
ADDITIONAL TASK GROUP INFORMATION
OBJD 0
OBJC 0
OVERLAY ZEBRA
LOCATION 1105
OBJA 0
OBJ6 0
OBJ5 0
OBJ1 0
TASK GROUP CONTROL STRUCTURES
ADDITIONAL TASK GROUP INFORMATION
OVERLAY FLOAT
OBJE 0
OBJD 0
OBJC 0
OBJA 0
OBJ6 0
OBJ5 0
OBJ1 0
TASK GROUP CONTROL STRUCTURES

Figure 4-5. Overlays in Memory Pool AA

FLOATABLE OVERLAYS CONTROLLED THROUGH OVERLAY AREAS

Only floatable overlays can be associated with overlay areas. Overlay areas are a mechanism that allows you to control the placement of floatable overlays without being required to write your own Overlay Manager.

As an example of overlay area use, assume that you desire to share both the root and overlays of a sharable unit whose structure is shown in Figure 4-6.

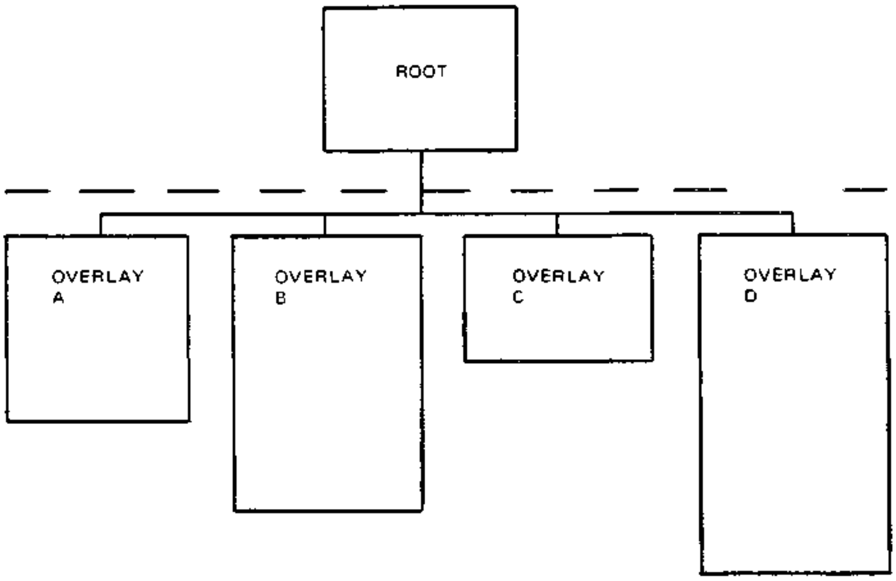


Figure 4-6. Sample Bound Unit Structure for Overlay Area Use

Assume further that Tasks 1, 2, and 3 (of the same or another task group) are executing the sharable bound unit and that Task 1 has encountered a create OAT function while executing the root.

When the create OAT function is encountered, an overlay area (controlled by the OAT) is created for the task group. In this example, the overlay area has three entries, each entry being 256 words long. There is no direct relationship between the number of overlays to be shared and the number of entries in the overlay area. The entries in an overlay area are of equal size. You must create overlay areas large enough to contain the largest overlay (overlay D in this example). The overlay area reserved is depicted below.

ENTRY 1	ENTRY 2	ENTRY 3
256 WORDS	256 WORDS	256 WORDS

When Task 2 (or Task 3) executes the same create OAT request (i.e., when it executes the root), the task is given the address of the OAT already existing in memory.

Assume that Task 1 issues a $OVRSV macrocall to reserve an overlay area defined by the OAT and to load overlay A in that area. The code and/or data composing Overlay A will be loaded in the first free overlay area; Task 1 will be given access to this area. At this instant the status of the overlay area is as follows:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 1	USAGE = 0	USAGE = 0
TASK 1		

When Tasks 2 and 3 now perform the request for Overlay A, they will be given access to the existing copy of the overlay. At this instant, the status of the overlay area is as follows:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 3	USAGE = 0	USAGE = 0
TASKS 1,2,3		

Task 2 now requests Overlay D. Since a task cannot have more than one overlay in an overlay area at any time, Task 2 must explicitly release Overlay A before requesting the loading of Overlay D. The result of releasing Overlay A and requesting Overlay D is as follows:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 2	OVERLAY D USAGE = 1	USAGE = 0
TASKS 1,3	TASK 2	

A request by Task 3 for Overlay C will result in the following situation:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 1	OVERLAY D USAGE = 1	OVERLAY C USAGE = 1
TASK 1	TASK 2	TASK 3

If there were another task in the group (e.g., Task 4) and the task were to request Overlay B, it would have to wait until one of the overlay areas was freed (by a Release Overlay macro call). If Task 4 requested Overlay A, C, or D, the task would be given access to the loaded copy of the overlay.

Note that at any given instant several OATs, controlling several different overlay areas, may exist. Even if a task is sharing overlays in different overlay areas, it cannot reference more than one overlay area at any given time. The task must release an overlay in an OAT prior to requesting an area for another.

UNLOADING OVERLAYS FROM OVERLAY AREA TABLES

You use a Release Overlay Area ($OVRLS) macro call to exit from an overlay. When this call is executed, the count of the number of users of the overlay is decremented in the defining OAT. When the count drops to zero, the overlay area is marked as available and can be reused by a Reserve Area and Execute Overlay function.

Overlay areas and defining OATs are deallocated when the last usage of a sharable bound unit has terminated.

LOADING BOUND UNITS (SEARCH RULES)

The Loader uses search rules to locate a bound unit to be loaded. The Loader starts the search in response to a command containing an argument naming the bound unit to be loaded.

Search rules that regulate the search process define three directory pathnames and the sequence in which they are used during a search. They are:

1. User task group working directory.

2. System directory -LIB1 argument of the Change System Directory (CSD) command.

3. System directory -LIB2 argument of the Change System Directory (CSD) command.

The operator command CSD can be used to change pathnames associated with system directory arguments -LIB1 and -LIB2. The pathname of a user task group's working directory is established through a CWD command or through the -WD argument in the Enter Batch Request (EBR), Enter Group Request (EGR), or SG commands.

TASK EXECUTION

Section 5
TASK EXECUTION

A task can be characterized as the execution of a sequence of instructions that has a starting point and an ending point and performs some identifiable function. In an Assembly language program, a task can initiate another task for execution or terminate itself by calling the task management functions. Multiple tasks can operate independently of and asynchronously to each other.

Each application, system, or device driver task operates at an interrupt priority level, one of the 64 priority levels provided by the hardware and firmware. This section describes the processing of priority levels, including context saving of interrupted tasks, and the assignment of priority levels and logical resource numbers to tasks. Communication between tasks, task coordination, and task handling by the system are also summarized in this section.

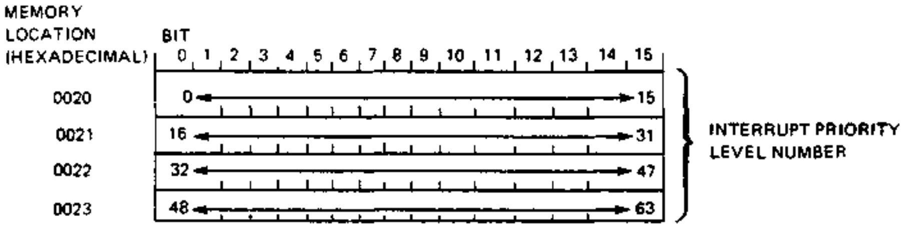
INTERRUPT PRIORITY LEVELS

All system tasks, device drivers, and application tasks are assigned interrupt priority levels that indicate the order of their execution. Control of the central processor is given to the highest active interrupt level. An overview of the priority levels, a description of how the hardware and firmware process priority levels, and information on controlling levels (the latter of interest primarily to the Assembly language programmer) are given below.

Processing Priority Levels

A DPS 6/Level 6 central processor provides 64 potential interrupt priority levels that are used by the hardware to order the processing of events. These levels are numbered from the highest priority (Level 0) to the lowest priority (Level 63). Levels 0 through 4 are reserved; Level 63 is the "system idle" level; the intervening levels (5 through 62) are assigned to logical resources (i.e., devices and tasks).

The determination of which priority level is to receive central processor time is based on a linear scan of the level activity indicators. The level activity indicators are maintained by the hardware in four contiguous dedicated memory locations (see Figure 5-1). Each bit that is "on" denotes an active priority level; each bit that is "off" denotes an inactive level.



NOTE IF THE BIT CORRESPONDING TO AN INDIVIDUAL LEVEL IS "ON" THAT LEVEL IS ACTIVE IF THE BIT IS "OFF" THE LEVEL IS SUSPENDED

Figure 5-1. Format of Level Activity Indicators

When a given priority level is the highest active level, it receives all available central processor time until it is interrupted by a higher priority level or until it relinquishes control by suspending itself (setting its level activity indicator off). If a priority level is interrupted by a higher priority level, its level activity indicator remains on and it will resume execution of the interrupted logical resource when it again becomes the highest priority level. Each time a priority level change occurs, the hardware/firmware saves the content of the previously highest active level and restores the context of the new highest active level. Interrupting a task, saving the context of a task, selecting and starting the highest priority level task, and restoring the context of a task are done without software involvement.

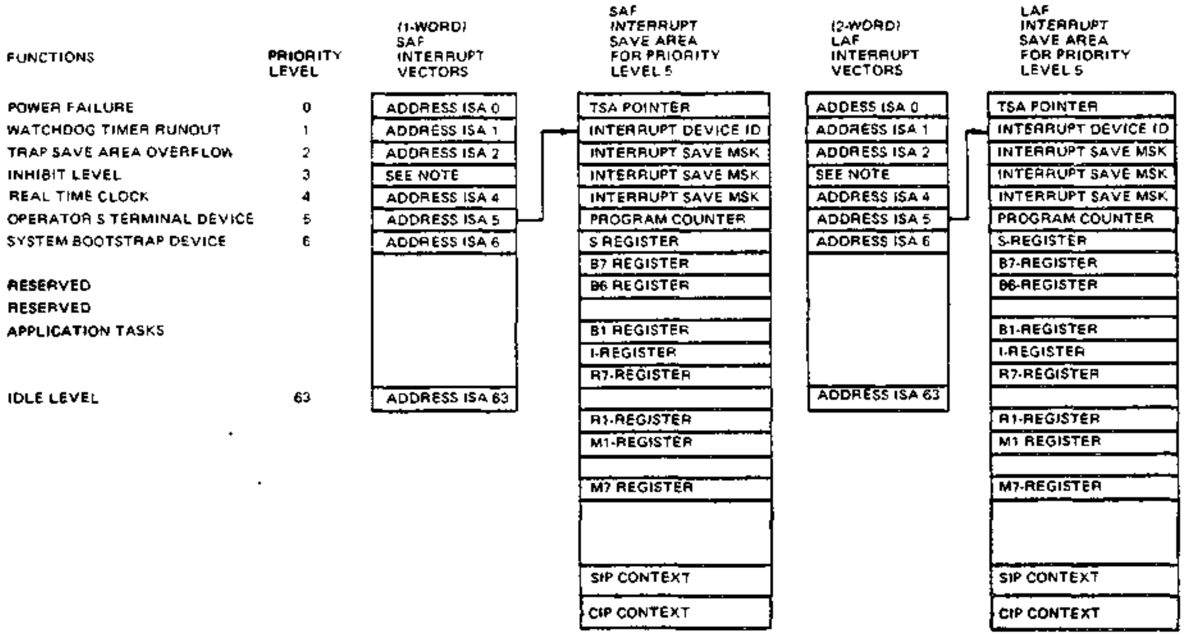
When more than one task is assigned the same priority level, software determines in round-robin fashion the next active logical resource to resume execution at this level. Thus a task does not block a level when the task is put in a Wait state after a request to wait, wait on list, request semaphore, terminate, or after a system service macro call that does a wait for a data transfer. Another task on the same level that is ready will be activated.

Timeslicing

An optional timeslicing facility can be configured. The timeslicing minimizes the ability of tasks that use large amounts of central processor time to interfere with interactive users of the system. Timeslicing is implemented as an extension of the real-time clock interrupt servicing task which executes at level 4. At each clock interrupt, it examines the tasks at the highest active user level. If the execution of that task exceeds a configured timeslice value without waiting for some event, then it is removed from the front of its level queue and placed at the end of the queue. If a separately configured number of such timeslices occur without the task's waiting on any event, then the task is demoted one priority level (i.e., the task's priority level is increased by one). The task can be demoted again and again until it has been demoted to priority level 62 or until a congifured maximum level is reached. Each time that a task that was demoted by the timeslicer waits for some event, it is elevated one level (i.e., its priority level is decreased by one) until the task reaches its assigned priority level. The timeslicing facility can be configured to perform timeslicing at a specified user level and below. Configuration of time slicing is discussed in the System Building and Administration manual.

Interrupt Save Area

The context of a level can include the contents of the program counter, the S-register, all B-registers, the I-register, all R-registers, all M-registers and all SIP and CIP registers. The context is stored in a block of memory known as an Interrupt Save Area (ISA). The hardware/firmware context save/restore function finds the appropriate ISA through a pointer supplied in the interrupt vector for that level. The interrupt vectors are a set of contiguous memory locations containing an entry for each potentially active priority level and ordered by ascending priority level number. Figure 5-2 illustrates the order of the priority levels, their corresponding interrupt vectors, and the format of an ISA.



NOTE The inhibit level (priority level 3) does not have its own ISA it points to the ISA of the priority level from which it was entered

Figure 5-2. Order of Interrupt Vectors and Format of Interrupt Save Areas

The three highest priority levels have dedicated assignments of special hardware/firmware functions such as incipient power failure, watchdog timer runout, and trap save area overflow. Priority Level 3 is reserved as an inhibit level and Level 4 is dedicated to the real-time clock. Succeeding levels are configured as device levels. Following these are two levels that are reserved for system use. Except for Level 63, the remaining levels can be used for application tasks. Level 63 is reserved for an always active software idle loop.

<u>Control of Priority Levels</u>

MOD 400 controls multiple levels through the use of the LEV (Level Change) instruction, which provides the following functions:

- Resume: Ascertain the highest active priority level by examining the level activity indicators. Restore the context of this priority level and continue on this level.

- Suspend: Mark the current priority level as inactive (reset the level's activity indicator), save the context of the current priority level, go into Resume state. Except for this description of a firmware state, in this documentation set the term "suspend" indicates the logical state of a task as described in the paragraph "Task Handling."

- Activate a Level: Mark the target priority level as active (set the level's activity indicator), save the context of the current priority level, go into Resume state.

- Inhibit: Mark dedicated, high-priority level as active and immediately assume this level. Continue execution on this priority level with no context save or restore.

- Enable: Return to the highest active normal priority level from the inhibit level. If the highest priority level is the same one from which the inhibit level was entered, do not perform a context save or restore.

The number of the highest active priority level can be ascertained by looking at the contents of the S-register.

The availability of an inhibit level allows critical sections of code to be executed without danger of interruption (e.g., it may be necessary to temporarily assume the inhibit level in order to protect short sequences of instructions that modify data structures shared between levels).

A standard feature of DPS 6/Level 6 central processors is a real-time clock, which interrupts at a preassigned priority level (Level 4) each time its specified scan cycle has elapsed.

TRAP HANDLING

The hardware provides a means by which certain events that occur during the execution of a task can be "trapped" with control passed to software routines that are designed specifically to cover the condition causing the trap. Events such as the detection of a program error, hardware error, arithmetic overflow, or uninstalled optional instruction cause traps (i.e., control transfers to designated software routines) to occur.

Traps fall into two classes: (1) standard system traps, for which routines are supplied with the system, and (2) user-specific traps, for which you can supply your own handler.

An application program can designate which traps are to be handled through specification of the enable/disable user trap macro calls (refer to the GCOS 6 MOD 400 System Programmer's Guide Volume II for details). If an enabled trap occurs in the user program, the Trap Manager transfers control to the connected trap handler for the condition causing the trap. A trap that is enabled is local to a task; such a trap neither affects nor is affected by the handling of the same trap in another task, even within the same task group.

Any trap that occurs when its handler is not enabled, or that does not have a handler to process it, causes the executing task to be aborted.

SYSTEM FEATURES AFFECTING TASK EXECUTION

MOD 400 does not monitor resource use either within a task group or among task groups using the online pools. Task and task groups must cooperate in their use of system resources to ensure smooth operation of the application.

Priority Level Assignments

Priority Levels 5 through 62 are available for assignment to system, device driver, and application tasks. The priorities of system tasks and driver tasks are established during configuration. The priorities of application tasks are assigned during task group creation. Priority levels with a low numeric value have a higher priority than those with a high numeric value. The procedures for establishing priorities are described below.

ASSIGNING PRIORITY LEVELS TO DEVICES AND SYSTEM TASKS

Priority levels for devices are established automatically as part of the interactive system building process (M4_SYSDEF). If you wish, you can directly specify priority levels through an argument of the Configuration Load Manager (CLM) DEVICE or DRIVER directive. When a particular type of device is specified in the M4_SYSDEF dialog or through a CLM DEVICE directive, the appropriate vendor-written device driver is loaded as part of the system. A CLM DRIVER directive is required only for a user-written driver. The two priority levels following the last one assigned to a configured device are used by system tasks and cannot be assigned to application tasks.

An example of priority level assignment is shown in Table 5-1. Levels 0 through 4 are assigned by the system and are not available to any user. The operator terminal is assigned Level 5 by M4_SYSDEF; you can assign any appropriate level to the operator terminal through a CLM DEVICE directive. At initialization, the system bootstrap device is assigned level 6. This assignment remains in effect unless it is changed by a CLM DEVICE directive. The $D Debug system program requires two priority levels. These must be higher than any task the $D Debug program is to debug. Thus for the assignments shown in Table 5-1, the $D Debug program must be assigned levels 7 and 8 if it is to be able to operate on the communications supervisor.

Table 5-1. Priority Level Assignments for Tasks and Devices

Physical Priority Level	Base Priority Level	Use	Comments
0 1 2 3 4	N/A N/A N/A N/A N/A	Power Failure Handler Watchdog Timer Runout TSA Overflow Inhibit Interrupts System Clock	Levels 0 through 4 are automatically assigned by the system.
5	N/A	Operator Terminal	Conventionally assigned Level 5, but can be assigned any available level.
6	N/A	System Bootstrap Device	Set to Level 6 at system initialization but can be changed.
7 8	N/A	$D Debug Program	Requires two levels. Can debug only those programs or drivers that have a lower priority (higher numeric value).
9	N/A	Communications Supervisor	Must be a higher level than any communications device.
10 10 10	N/A N/A N/A	TTY Device TTY Device TTY Device	Communications devices can share priority levels.
11 11	N/A N/A	Removable Cartridge Disk Fixed Cartridge Disk	The priority level for a pair of fixed/removable disks must be the same.
12 13 14	N/A N/A N/A	Diskette Diskette Diskette	

Table 5-1 (cont). Priority Level Assignments for Tasks and Devices

Physical Priority Level	Base Priority Level	Use	Comments
15	N/A	Line Printer	
16	N/A	Card Reader	
17	N/A	Reserved by System	The next three levels following the last used for a configured device are used by the system.
18	N/A	Reserved by System	
19	N/A	Reserved by System	
	0	Task Group A	
	1	Task Group B	
	. . .	. . .	
	10	Task Group n	
. . .			
63	N/A	System Idle Loop	Always active.

Table 5-1 indicates Input/Output (I/O) devices, and not device drivers, to stress that each peripheral device must have at least one level assigned to it; peripherals (other than communication devices) cannot share a level. If there are two printers, each must be assigned a unique level even though there is only one copy of a reentrant I/O driver. Communications configurations require at least one nonsharable level dedicated to processing communications interrupts; it must be at a higher level than any communications device. Communications devices can share a level. For example, four Teleprinters (TTYs) and one Visual Information Projection (VIP) can be configured to share one level or to use up to five levels. The priorities in Table 5-1 provide maximum throughput because high-transfer-rate devices are assigned a higher priority than low-transfer-rate devices.

Theoretically, you could assign a level number as high as 59 to a device. In which case, Levels 60 and 61 would be used by the system and Level 62 would be assigned to a user task group. In practice, however, you would want to reserve levels for more than one user task group, especially for a system with a large number of devices. If priority Levels 5 and 6 are assigned as shown in Table 5-1, the theoretical range of levels assignable through CLM COMM directives is 7 through 58. For a device associated with a COMM directive, the range is 8 through 59.

ASSIGNING PRIORITIES TO APPLICATION TASKS

Priorities are assigned to user task groups and tasks when they are created or spawned. The command to generate a task group contains an argument that specifies the base priority level for the task group. The base priority level is relative to the highest number of the priority level that has been assigned a configured device. When a task group is assigned a base priority level of zero, the lead task of the group executes at the physical interrupt priority level that is three level numbers above the highest level number assigned to a configured device. When other tasks in the same task group are created or spawned, they are given level numbers relative to the base priority level assigned to the task group. The physical interrupt level at which a task executes is the sum of the following:

1. The highest level number assigned to a configured device plus 3

2. The base priority level number of the task group.

3. The relative priority level of the task within that group.

This sum must not exceed 62. Tasks that have the same base priority level are processed on a round-robin basis.

User tasks that execute online are usually given higher priorities (lower level numbers) than those executing in batch. Tasks that are I/O bound should be run at a higher priority than tasks that are central-processor bound. This permits I/O-bound tasks, which run in short bursts, to issue I/O data transfer orders as needed, wait for I/O completion, and, while in the Wait state, relinquish control of the central processor to the central-processor-bound tasks. Otherwise, if the central-processor-bound tasks have a higher priority, the I/O devices would be idle while I/O-bound tasks waited to receive central processor time. The optional timeslicer can be used to minimize the ability of central-processor-bound tasks to interfere with interactive and I/O-bound tasks.

Logical Resource Number

A Logical Resource Number (LRN) is an internal identifier used to refer to task code and devices independent of their physical priority levels. Use of LRNs makes Assembly language application task code independent of priority levels so that, if circumstances require a change in priority levels, the task code does not have to be reassembled.

DEVICE LRNs

LRNs are automatically assigned to devices when the system is configured through the interactive system generation process. If desired, you can use the CLM DEVICE directives to assign your own LRN values.

Figure 5-3 is an example of priority level assignments for devices and system tasks and the related device LRNs.

LRN	LEVEL	
	0	
	3	INT
	4	CLOCK
0	5	OPERATOR'S TERMINAL
1	6	DISK
3	7	LINE PRINTER
4	8	SERIAL PRINTER
5	9	CARD READER
	10	OPERATOR INTERFACE MANAGER INTERRUPT
	11	SYSTEM TASK

Figure 5-3. Example of LRN and Priority Level Assignments to System Tasks and Devices

APPLICATION TASK LRNs

LRN assignments to application program tasks are not dependent on the system configuration on which the application task group is running. You can either assign LRNs or have the system select the highest numbered LRN available at task creation. LRNs are assigned to task code within an Assembly language application program through specification of the Create Group and Create Task macro calls, as well as the macro calls that build data structures ($IORB, $TRB, etc.). LRNs can be assigned at the control lan- guage level through the use of the commands (including operator commands) for creation of tasks groups and tasks. An LRN for an application task can have any value from 0 through 252. Within a task group, the LRN for each task must be unique. More than one LRN can be associated with the same level (i.e., two tasks at Level 23 can be assigned LRNs of 28 and 29, respectively).

Two kinds of tasks do not have LRNs:

- The lead task of any task group
- Any spawned task.

Logical File Numbers

Logical File Numbers (LFNs) are internal file identifiers that are associated with file pathnames at the Assembly language level or at the command level, through Create File (CR), Get File (GET), or Associate (ASSOC) commands. LFNs can be used to reduce program dependence on actual file pathnames, which are likely to vary.

Task and Resource Coordination

Tasks can be coordinated in either of two ways:

- Through the use of tasking requests
- Through the use of semaphores.

TASK REQUESTS

One task can request another to execute asynchronously with it, or the requesting task can later wait for the completion of the requested task. Both tasks have access to the request block provided by the requesting task and thus can pass arguments between them.

SEMAPHORES

Semaphores support an application-designed agreement among tasks to coordinate the use of a resource such as task code or a file. A semaphore is defined by a task within a task group and is available only to the tasks within that group.

For each resource to be controlled, a semaphore is defined and given a two-character American Standard for Information Interchange (ASCII) semaphore name. This name is a system symbol recognized by the system control software, not a program symbol that needs Linker resolution. The agreement is that each requestor of a resource whose use must be coordinated issues appropriate system service macro calls to the named semaphore to request or release the resource. The task that defines the semaphore assigns the semaphore's initial value. The system control software maintains its current value to coordinate requestors of the resource being controlled. A requestor obtains use of a resource if the semaphore value is greater than zero at the time of the request. A requestor is either suspended (waiting for the resource) or notified that no resource is available if the value is zero or negative.

System service macro calls are used to:

- Define a semaphore and give an initial value ($DFSM).

- Reserve a semaphore-controlled resource ($RSVSM); this macro call subtracts a resource or queues a waiter for the resource (i.e., it decrements the current-value counter).

- Release a semaphore-controlled resource ($RLSM); this macro call adds a resource or activates the first waiter on the semaphore queue (i.e., it increments the current-value counter).

- Request the reservation of a semaphore-controlled resource ($RQSM); this macro call queues a request block (SRB) if the resource is not available. This macro call decrements the current-value counter.

- Delete a semaphore ($DLSM).

A semaphore is a gating mechanism, and the initial value given to it depends upon the type of control you want to exercise.

For example, assume that you want to restrict access to a particular resource to a one-user-at-a-time order. The mechanism would work in the following way:

1. Task A defines a semaphore by issuing the macro call:

   ```
   $DFSM ZZ
   ```

   Omission of the value argument causes the initial value to be set at 1.

2. Task B now issues a $RSVSM call; the counter is decremented to 0. Task B gets the resource for itself, knowing that no other task using the semaphore mechanism is using or can obtain the resource.

3. Task C issues a $RSVSM call; the counter is decremented to -1. Task C is suspended and put on the semaphore queue in First-In/First-Out (FIFO) order (Task B is still using the resource).

4. Task B issues a $RLSM call when it finishes with the resource; the counter is incremented to 0. Task C now gets the resource. After the $RLSM call for Task C, the value is 1 again.

Use of resources by more than one user at a time can be arranged by adjusting the initial value of the semaphore (e.g., an initial value of 2 allows two users, a value of 4 allows four users, and so on, depending on the nature of the resource and its intended use).

If it is undesirable for a task to be suspended while a resource is in use, the $RQSM macro call can be used instead of $RSVSM to reserve a resource. $RQSM is an asynchronous reservation request ($RSVSM is a synchronous request) which causes a request block to be queued for the resource, so that the issuing task can do other processing before the needed resource is available.

TASK HANDLING

More than one task can be concurrently active under MOD 400. In a multiprogramming environment, a task in each of several task groups can be active and compete for system resources. Another possibility is a multitasking application where several tasks executing under one task group can be active to compete for system resources among themselves and with tasks from other task groups. A COBOL, BASIC, or RPG program executes as a single task. A FORTRAN or Assembly language program can include requests to activate several tasks and synchronize their execution; these requested tasks can execute concurrently.

In order for the system to sequence the execution of tasks, each task must be assigned to a priority level. Task competition for the central processor resource is governed by the hardware/ firmware linear priority scan of level activity indicators. Tasks on the same priority level execute serially in the order in which they are requested. The highest priority active task receives all available central processor time until it is interrupted by a higher priority task or until it waits, terminates, or is placed in hold. As a result of this linear ordering of task priority, care must be taken that high-priority tasks do not consume an excessive amount of central processor time at the expense of lower priority tasks. A task that has a built-in program loop to wait for an event occurrence prevents other tasks at the same or lower priority levels from executing.

In an Assembly, COBOL, or FORTRAN language program, program loops might not be necessary, since a Wait function can be invoked by a task (at some point after a related request has been made) to suspend itself and to be reactivated later (at the time of event completion).

It should be noted that all device drivers are considered to be tasks in the above sense; using the File System, buffered device drivers can execute concurrently with tasks. Drivers execute on the priority levels assigned to individual devices and thus have their own contexts. The device drivers provided in the system are written in reentrant code and are therefore capable of servicing multiple devices.

A user task becomes active when a Spawn Task (ST) or Enter Task Request (ETR) command is issued for it. The ST command can request that the invocation of the task be delayed until a specified time interval has elapsed. FORTRAN programs can cause a task to become active through the START and TRNON statements. Assembly language programs can issue a $RQTSK or $SPTSK macro call to activate a user task. The $SPTSK macro call can specify that the task be invoked only after a certain period of time has elapsed.

Tasks can exists in any of the following logical states:

- Dormant: There is no current request for the task. A task enters the Dormant state if it is created but never requested, or a terminate request is issued against it. A task remains Dormant until a request is placed against it or it is deleted. If deleted, it is erased, memory is deleted, and it cannot be reactivated.

- Active: A task is executing or ready to execute when its priority level becomes the highest active level in the central processor. A task remains active until it waits, terminates, or is suspended. In the Active state, task execution might stall by not having task code executed (e.g., the task issues a synchronous I/O order).

- Wait: A task is not executing because it may have caused its own execution to be interrupted until the completion of an event such as the completion of a requested task, or until a timer request is satisfied, or until a task releases a semaphore. A waiting task loses its position in the priority level round-robin queue. An I/O order to disk, magnetic tape, operator terminal, or unbuffered card reader always results in a Wait condition. Task code written in FORTRAN or Assembly language will also wait in the following circumstances: a write to an interactive terminal or to a printer when a previous write has not completed; a read order issues before the transfer of the current message from an interactive terminal is complete (i.e., RETURN key not pressed). In COBOL, the latter two circumstances result in a Wait if the program is executing its I/O statements in synchronous mode; otherwise, if in asynchronous mode, the result is a status return code value of 9I with no waiting.

- Suspend: A task is removed from execution by an external human action (e.g., the operator enters a Suspend Group (SSPG) command or a user interrupts a program with a Break). The task is activated through another human action (e.g., the operator enters an Activate Group (ACTG) command or a user enters a command after a Break).

To terminate, tasks of Assembly language programs must contain a Request to Terminate ($TRMRQ) call. Compilers provide this call in the object text. $TRMRQ is executed after the user completes execution.

When you desire the concurrent execution of more than one task, each task is specified in a Create Task (CT) or ST command or system service macro call.

The procedural code for a requested task is either in a unique bound unit or shared with a bound unit of a task that was previously created. When a task is requested, the system control software searches the table of LRNs of the current task group under which the task is executing for the identifying LRN and activates the task if it is not already active.

EXAMPLE OF SYSTEM INTERACTION WITH USER TASKS

The following sequence of events illustrates a typical interaction between the system control software and two tasks within a group. For the purpose of this example, Task A has an absolute priority level of 13 and Task B an absolute priority level of 12. The absolute priority level is obtained by adding a task's relative priority level, the task group's base priority level, and the highest system physical priority level plus three.

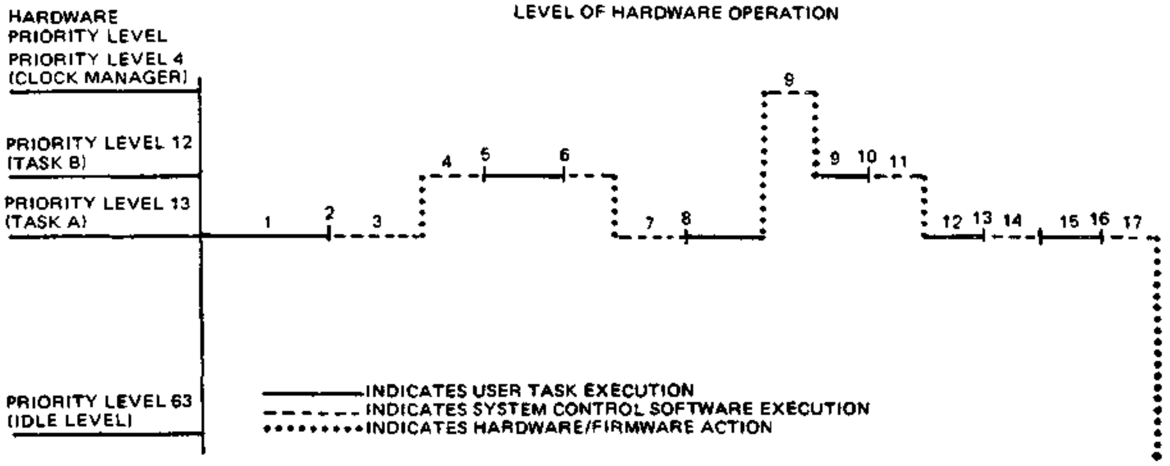
Figure 5-4 indicates the priority levels at which the central processor runs as the sequence of events occurs. The diagram also indicates the consecutive activity of user tasks, the system control software, and the hardware and firmware. The numbers in the diagram correspond to the numbers in the sequence of events and are explained in order in the text.

INTERTASK AND INTRATASK GROUP COMMUNICATION

Information can be passed among task groups and tasks by means of request blocks, common files, and the message facility.

Request Blocks

Task code written in Assembly language can pass information to other Assembly language tasks in the same task group by using variable-length request blocks. The request blocks can contain data or pointers to information structures. All request blocks must be in common address space so that they can be shared by the tasks. (Refer to the System Programmer's Guide for details on building request blocks.) Higher level languages cannot use request blocks directly; they require called subroutines written in Assembly language.


HARDWARE PRIORITY LEVEL
LEVEL OF HARDWARE OPERATION
PRIORITY LEVEL 4 (CLOCK MANAGER)
PRIORITY LEVEL 12 (TASK B)
PRIORITY LEVEL 13 (TASK A)
PRIORITY LEVEL 63 (IDLE LEVEL)
1
2
3
4
5
6
7
8
9
9
10
11
12
13
14
15
16
17
——— INDICATES USER TASK EXECUTION
– – – – – – INDICATES SYSTEM CONTROL SOFTWARE EXECUTION
••••••••• INDICATES HARDWARE/FIRMWARE ACTION

User Task Execution	System Control Software Execution
1. Task A is running; task B is requested by task A, or entered or spawned through a command.	
2. Task A does not issue a wait.	
	3. Places the request in the request queue for task B. (Assume that there are no other requests in this request queue.) Activates priority level 12.
	4. Examines the request and ascertains task B's starting address from start address data accompanying the request.
5. Task B begins execution because its priority level is higher than that of task A.	
6. Task B issues a call to the clock manager and issues a wait function to wait for clock timeout. Task B is suspended.	
	7. Now operating at the priority level of task A, the highest active priority level, returns control to task A.
8. Task A resumes execution.	
9. Task B's clock-related wait times out. The clock manager interrupts task A. Task B's priority level is activated. Task B resumes execution and continues to completion.	
10. Task B issues a terminate call.	
	11. Removes the request from the request queue for task B. Suspends priority level 12.
12. Task A resumes execution because its priority level is now the highest active level.	
13. In a multitasking program, task A could issue a wait call to wait for completion of task B.	
	14. Detects that task B's request is marked as terminated. Control is immediately returned to task A.
15. Task A continues to completion.	
16. Task A issues a terminate call.	
	17. Removes the first request from the request queue for task A. If there are no additional requests in this request queue, suspends priority level 13. If there are no remaining active priority levels, idles at priority level 63.

Figure 5-4. System Interaction with User Tasks

Common Files

Tasks within the same task group and tasks within different task groups can communicate via disk files. The concurrency status must be the same for all tasks using the files. The requesting tasks must have access rights to the files.

Message Facility

The message facility allows two or more task groups (users) or two or more tasks within a task group to communicate with one another. This communication is done through containers called "mailboxes." Messages (requests) sent to a task or task group are queued in a mailbox and are dequeued when received.

To control the sending and receiving of messages, the message facility provides a number of macro calls and commands. One set of macro calls (Initiate, Send, and Terminate Message Group) allows a message (a request) to be sent to a mailbox; another set of macro calls (Accept, Receive, and Terminate Message Group) allows a message to be received from a mailbox. Commands are provided to send, receive, list, and cancel messages (requests). A set of Send/Receive Message commands allows you to send messages (mail) to another user's mailbox and to display mail in your own mailbox.

Deferred processing of print, punch, and task group requests is carried out through the use of the message facility. Deferred processing is described in Section 6.

Before the message facility commands or macro calls can be used, and before the deferred processing of print, punch, and task group requests can be initiated, you must create the mailboxes and activate the message facility task.

The paragraphs below describe mailbox creation, the activation of the message facility task, and the command and macro call interfaces. Section 6 describes the deferred processing facilities.

CREATING THE MAILBOXES

Three steps must be performed to construct a mailbox. The user (or operator) must create the mailbox root directory on the local volume, create the mailboxes, and set access controls on the created mailboxes. (Refer to the Commands manual for details.)

The mailbox root directory is the directory that is to contain the simple names of the mailboxes.

The system assumes that the mailbox root directory is in the MDD directory. (An MDD directory is supplied by the vendor on the system root volume). You, however, are free to create your own mailbox root directory through the standard Create Directory command.

Each mailbox is created through the Create Mailbox command. This command creates a directory corresponding to the mailbox name and a file ($MBX) within that directory defining the mailbox attributes.

To prevent unauthorized use of the message queues, the user or operator should set access controls as follows:

- Senders must be given list access on the directory defining the mailbox.

- Receivers must be given read access on the $MBX file for a given mailbox.

Individual mailboxes can be deleted through Delete Mailbox commands.

ACTIVATING THE MESSAGE FACILITY TASK

The Start Mail operator command activates the message facility. The Start Mail command contains an optional argument used to set the name of the mailbox root directory to other than the default directory pathname (MDD).

MESSAGE FACILITY COMMAND INTERFACE

The commands that can be used to send/receive messages (mail) are Mail (MAIL), Send Message Mailbox (SMM), and Accept Message Mailbox (AMM). Commands are also provided to list and delete messages.

The MAIL command is used to send and receive multiline messages to/from the mailbox whose name (id) is the same as the person id of the receiving user. A message sent by a MAIL command is queued in the mailbox and displayed only if the receiving user issues a MAIL command.

To send messages, issue the MAIL command, specifying the mailbox id (person id) of the user to receive the messages. The messages to be sent can be located in a file (named by an argument of the command) or they can be entered after the MAIL command has been executed. You can use the -TIME argument in the MAIL command to defer the displaying of messages until a specified date and time.

To receive messages, issue the MAIL command without arguments. The contents of your mailbox are displayed when the command is executed. If you request deletion of the messages, they are deleted from the mailbox after being displayed. Otherwise, the messages remain in the mailbox. Messages whose date and time for display have not been reached are not displayed.

The SMM and AMM commands are used for short messages that must be viewed immediately or at a specified time. The AMM command is used to specify that messages sent by the SMM command be displayed when received or at the designated time.

To send messages, issue the SMM command, specifying the person id (mailbox) to which the messages are to be sent. The message is included in the command line. You can use the -TIME argument to specify a delivery time for the message.

To receive messages, issue the AMM command, specifying the mailbox from which you wish to receive messages. Once this command is issued, messages are displayed when they are placed on the named mailbox by the SMM command. The messages are deleted from the mailbox as soon as they are displayed. Messages whose date and time for display have not been reached are not displayed.

You can send alternate MAIL and SMM commands. A receiving user who issues a MAIL command receives both types of messages. A user who issues the AMM command receives only messages sent by the SMM command.

The -IMBX argument of the AMM command allows you to specify by name the sending user from whom you will accept messages for immediate display. Messages sent to you by other senders are stored in your mailbox. The -AMBX argument of the AMM command allows you to specify the name of a mailbox (other than your own) whose messages you wish to be displayed to you. By using the -AMBX argument, you receive someone else's mail and have it displayed when it is sent.

Through the proper combination of SMM and AMM commands, you can achieve a broadcasting facility.

MESSAGE FACILITY MACRO CALL INTERFACE

You can use the message facility on an Assembly language level by using the macro call interface. For Send and Receive messages, the message facility provides the following macro calls: Initiate Message Group ($MINIT), Send ($MSEND), Accept Message Group ($MACPT), Receive ($MRECV), Terminate Message Group ($MTMG), Count Message Group ($MCMG), and Cancel Enclosure Level ($MCME). The information associated with these macro calls can be passed by using the appropriate request blocks.

A task group that wishes to send a message to the mailbox must issue an $MINIT macro call to open the Send message session. The mailbox is identified by a name entered in the request block. As a result of this macro call, the message facility returns a message id, unique to the task group, to identify the message to the other macro calls (i.e., send). The task group then issues one or more $MSEND macro calls to send message data. The Send message session is closed by the $MTMG macro call or alternatively, by the $MSEND macro call. The sending task group can issue the $MCME macro call to delete the last record of an incomplete quarantine unit or the entire incomplete quarantine unit. Receipt of the message can be deferred by the sender.

A task group wishing to receive a message from a mailbox issues an $MACPT macro call to open the Receive Message session. The mailbox is identified as described above for the $MINIT macro call, and the message facility returns a message id to be used by the $MRECV and $MTMG macro calls.

The task group then issues one or more $MRECV macro calls to receive message data. The Receive Message session must be closed with a $MTMG macro call.

The message may be accepted on the following selection criteria: first available, sequence number, submitter name, or submitter name and sequence number.

The receiving task group can request the message in record sizes other than those in which the message was sent. The receiving task group delimits the amount of received data by range or enclosure level.

The message facility can be used most effectively by two task groups wishing to communicate if they both simultaneously send and receive a message. To accomplish this, each of the task groups should issue the $MINIT macro call to open the Send Message session and the $MACPT macro call to open the Receive Message session. In this case the quarantine unit is a subject to exchange data between the two task groups.

[illegible] PROCESSING
CAPABILITIES

Section 6
DEFERRED PROCESSING CAPABILITIES

MOD 400 supports various facilities to defer processing. These deferred processing facilities are supported by the Message Facility (refer to Section 5). In deferred processing, the messages are requests.

Deferring the execution of group and batch processing requests makes it possible for you to gain greater control over the processing sequence. Deferring print and punch requests allows you to obtain program independence from the availability of print and punch devices. Queueing and later transcribing reports provides a spooling capability that places printing and punching outside of program context.

DEFERRING BATCH AND INTERACTIVE GROUP REQUESTS

When placing a batch or interactive task group request, you can have the request entered in a disk queue and can postpone any action being taken on the request until a specified time. When the request queue structures are on disk, memory space is conserved and the data in the queues can be recovered in the event of a system failure (refer to Section 7).

Two steps are required to defer group requests. The operator must create the request queues (mailboxes), and you must issue batch or interactive group requests with an argument specifying the time the request is to be activated.

Creating Group Request Queues

The operator must use the Create Group Request Queue command to create queue structures in which the group requests will be stored. The operator must also issue a Start Mail command if one had not been previously issued. These procedures are described in the System User's Guide.

Queuing Group Requests

You queue batch or interactive group requests by issuing an Enter Batch Request or Enter Group Request command, respectively. You can postpone action being taken on the request by specifying the -DFR (defer for interval) or -TIME (defer until date/time) arguments.

Once the operator has issued a Create Group Request Queue command for the task group, all further group requests are queued whether or not the requests are being deferred.

If the operator does not issue a Create Group Request Queue command, you can still submit group requests but will not be able to defer the requests.

DEFERRING PRINT AND PUNCH REQUESTS

The system provides a deferred printing and punching capability under which your requests for printing or punching specified files are queued in memory or disk mailboxes. The actual transcription of the files is done at a later time under the control of a system task group called a Daemon.

After you submit a deferred print or punch request, you can resume normal activities, log off, or rebootstrap the system without losing the request.

The three steps involved in deferred print and punch processing are creating the mailboxes, activating the daemon, and queuing the print or punch requests. The information in the following paragraphs is conceptual. Procedures for deferred printing and punching are in the System User's Guide.

Creating Print and Punch Request Mailboxes

The operator establishes the mailboxes that are to contain the queued print or punch requests. The mailboxes can be in memory or on disk. The mailbox names must be in the form $PR.Qn for mailboxes used to contain print requests and $PU.Qn for mailboxes used to contain punch requests (n is an integer from 0 through 9 that identifies the relative priority of the queue, with 0 being the highest priority and 9 the lowest).

Creating the Print and Punch Daemon

The operator is responsible for defining and activating the Daemon to process the print and/or punch requests. A print Daemon, a punch Daemon, or a print/punch Daemon can be created. (The supplied START_UP.EC file creates a standard Daemon task group at system startup. The operator can accept this Daemon, modify it, or create his/her own.)

To create a Daemon task group, the operator issues a Start Mail command (if one was not already issued), a Create Group command naming the Daemon to be created, and an Enter Group Request command identifying the mailboxes to be used for queuing the requests and the devices to be used for printing or punching.

Multiple Daemon task groups can be run concurrently using common or separate sets of mailboxes, printers, and punches.

Queuing Print and Punch Requests

Once the Daemon task group is active, you can queue print or punch requests by issuing Deferred Print and Deferred Punch commands. You can employ the -TIME argument in these commands to defer the printing or punching of a file until a specified date and time.

QUEUING AND TRANSCRIBING REPORTS

Any file in print or punch format (i.e., any report file) can be queued and subsequently transcribed to an available printer or card punch. Report queuing and transcription is a spooling capability that provides automatic and manual report transcription, time-of-day printing or punching, and an automatic setup function that includes a sample transcription file (template).

The report queuing and transcription facilities control report transcription outside the context of the program. Reporting procedures for identical software can be totally different in different situations without requiring reprogramming.

Report queuing and transcription have three major aspects: creating a report queue, queuing a transcription request, and transcribing a report.

Creating Report Queues

A report queue is a directory that allows you to place a report in a queue and subsequently transcribe the report. Report queues are created, modified, and deleted through Report Queue Maintenance (RQM) commands. The characteristics of the report queues are determined when the queue is created; the contents are determined when a report is placed in the queue for later transcription.

When the report queue is created, a report queue profile file is built. The report queue profile file designates the characteristics of reports that will be entered in the report queue and printed or punched at a later time. The report characteristics include:

- Name of form descriptor
- Format of reports to be queued (print or punch)
- Transcription mode (automatic or manual)
- Column number at which printing is to begin
- Line at which printing is to begin (head of form)
- Number of print lines per inch
- Number of copies of report
- Time at which report is to be transcribed
- Heading line
- Destination line.

The report queue profile file is complete when the report queue is created; however, various aspects of the profile can be overridden when the report is queued.

Queuing Report Requests

The name of a report to be subsequently printed or punched is placed in a report queue through the Queue Report (QRPT) command. This command also associates with the report a specialized report queue profile file that governs the details of the report transcription. Once a request has been queued, it remains queued until the file has been transcribed or the request pathname has been deleted through a report queue maintenance renew or delete function.

Transcribing Reports

Previously queued reports are written to a printer or card punch through the Unspool (UNSP) command. A single UNSP command can unspool all current and future reports. The printing or punching characteristics are determined by the report queue profile file created through the RQM command, the specialized report queue profile file created by the QRPT command, the user's activities, and the arguments specified in the UNSP command.

The UNSP command defines the report queue and the hard copy device to be used. After the command is executed, the specialized report file (if any) is deleted from the report queue. All reports whose profile matches the specified profile are unspooled in a single invocation of UNSP.

The report queue profile file can specify that the report is to be transcribed automatically or manually.

Automatic transcription is used when constant monitoring of a report queue is desired. When there is no transcription activity in progress, the unspool routine suspends itself for 1-minute intervals. When transcription of the queue is activated, each report in the queue is printed immediately unless one of the following is true:

- Manual mode was specified in the controlling profile.
- The specified time of day for report transcription has not been reached (or exceeded).

Manual mode is used to print reports in a nonautomated fashion. When the reports are required, the UNSP command is issued. All reports on the queue are transcribed immediately, regardless of time or mode. When the print queue is empty, UNSP terminates.

BACKUP AND RECOVERY
FACILITIES

Section 7
BACKUP AND RECOVERY FACILITIES

MOD 400 supports facilities that enable you to save and restore disk files, preserve the execution environment during a power failure, perform file recovery at the record level, and restart a program from a previously established point.

The Save/Restore facility provides you with the capability to preserve selected disk files and directories on magnetic tape or another disk volume and, when later required for processing, to restore the files, directories, and associated structures to disk.

The Power Resumption facility uses the memory save and auto-restart unit to preserve the memory image through a power failure lasting up to 2 hours. If power is restored during this time, the Power Resumption facility reconnects the previously online peripheral and communication devices and restarts the tasks that were running when the power failure occurred. If the power failure lasts more than 2 hours, the memory image is destroyed and the Power Resumption facility disabled. When power is restored, you can reinitialize the system and use the File Recovery and Checkpoint facilities to restart the system from a previously established restart point.

File Recovery enables you to dynamically save record images before they are updated and, if necessary, later write the images back to the file, thereby returning the file to its unaltered state. It provides file integrity in the event of a system failure and is provided through three distinct functions:

- Before-Image recording: Preserves a record prior to its being updated.

- Cleanpoint or Checkpoint declarations: Issued in the user's program and define a point at which all updates are complete. When the updates are complete, the associated before images are destroyed.

- Rollback, Recovery, or Restart functions: Return the files to their unaltered state by applying all before images that have been recorded since the last cleanpoint.

File restoration procedures enable you to reconstruct disk files and/or volumes that are damaged as a result of a device failure. File restoration is provided through two distinct functions:

- After-Image recording: Preserves a record of the updates made to files.

- Roll-Forward Utility: Reapplies updates (after images) to files to bring them up to their most recent consistent state before the device failure.

After images are used in conjunction with the Save, Restore, and Roll-Forward utilities to return files to a known state if data in the files is destroyed as a result of a device failure.

The cleanpoint, rollback, and recovery functions should be used to provide file recovery in a transaction-oriented environment. They are best suited for applications in which a single transaction causes a number of record updates. In a batch processing environment, the checkpoint and restart procedures should be used for file recovery and program restart.

The Checkpoint Restart facility enables you to establish a point in your program to which you can return at a later time and continue processing. The return point (checkpoint) is used to save the current status of the task group. You issue a checkpoint call in your program when you reach a point in your processing at which the program could be restarted. A restart can be performed at the most recently completed checkpoint at any time during processing. If the task group is abnormally terminated for any reason, it can be restarted at the most recent valid checkpoint.

DISK FILE SAVE AND RESTORE

The Save and Restore commands allow you to save and restore disk files and directories. Save is used to save disk files and directories on a disk or magnetic tape volume for later restoration by the Restore command.

The Restore command reconstructs the file structures copied by the Save command. If a file being restored already exists on the volume (or volumes), the Restore command replaces the current file contents with the file data saved by the Save command. (The access control list is not altered.) If a file being restored does not exist on the volume, the Restore command creates the file and loads the saved data. (Access is set as defined in the saved file.)

POWER RESUMPTION

Power Resumption is an optional facility that allows the system execution environment to be automatically restarted after a power interruption. The DPS 6/Level 6 central processor must have the memory save and autorestart unit. This unit can preserve the memory image through a power failure lasting up to 2 hours. (It cannot, however, preserve the state of the I/O controllers nor ensure that no operational changes have been made to the mounted volumes.)

If fewer than 2 hours have elapsed when power is returned to the central processor, the Power Resumption facility will perform the following functions:

- Reinitialize the system software.

- Reconnect peripheral devices.

- Reconnect communication devices serviced by the Asynchronous Terminal Driver (ATD) line protocol handler or the Teleprinter (TTY*) line protocol handler. (Refer to the System Building and Administration manual and the System Programmer's Guide for information about these line protocol handlers.)

- Restart application tasks that were active at the time of the failure, if these are display formatting and control facility tasks or are tasks containing user-written code to handle power failure and power resumption.

Implementing the Power Resumption Facility

The Power Resumption facility must be included in the MOD 400 Executive at system building. The DPS 6/Level 6 central processor must contain a memory save and autorestart unit that has been activated by the operator (refer to the System User's Guide for activation procedures).

When Power Resumption is specified in the system building dialogue, all peripheral devices and all communication devices associated with the ATD and TTY line protocol handlers are designated as reconnectable and will be automatically reconnected when power is restored. If any ADT or TTY* associated device is not to be automatically reconnected, the system builder must edit the

CLM file to remove the -RECONNECT argument from the Set Terminal Characteristics (STTY) directive generated for the device.

Power Resumption Functions

The Power Resumption facility automatically performs the following functions:

- Restarts the device drivers, clock, communications subsystem, and display formatting and control facility.
- Reconnects all peripheral devices that were online at the time of the failure.
- Reconnects ATD or TTY*-associated communication devices that were online at the time of the failure, except for those devices designated as not reconnectable.
- Restores the screen forms on reconnected terminals controlled by the display formatting and control facility.
- Resets the system date and time if the date/time clock has a separate battery backup unit.
- Reloads the Memory Management Unit (MMU), if any is present.
- Reestablishes the integrity of mounted volumes.
- Restarts application tasks that were active when the power failure occurred, if they are display formatting and control facility tasks or tasks containing user-written code to handle power failure and power resumption.

If an application task is to be notified when a power resumption has occurred, it must be written to check Trap 53 when the task becomes active and is issuing its own instructions (not executing Executive instructions). (Refer to "Trap Handling" in Section 5.)

After a power resumption has occurred, peripheral devices and reconnectable ATD or TTY* associated devices that were online at the time of the failure are again brought online. The operator may be required to initialize certain peripheral devices. A terminal user may be required to reenter the input line if he/she had not pressed the RETURN or XMIT key when the failure occurred. (Refer to the System User's Guide for details.)

FILE RECOVERY

The File Recovery facility enables you to save record images from a file before it is updated and to later write these images back to the file, eliminating the alterations made during the updating. Every time a record is updated, a copy of the record, as it exists before the update, is written to a system-created file. The system-created file is called a recovery file; the records it contains are called before images. The system uses the recovery files to bring your data files to a consistent state following a software failure or a system failure such as that caused by a loss of power. When the before images are applied in reverse chronological order to your data files, the data files are rolled back to a previously established state.

Designating Recoverable Files

File recovery is optional. You designate a file as recoverable through the -RECOVER argument of the Create File (CR) command. Files not created as recoverable can be made recoverable by specification of the -RECOVER argument of the Modify File Attribute (MFA) command. Recoverable files can be made nonrecoverable through the specification of the -NORECOVER argument in the MFA command.

Recovery File Creation

Each task group (or task in some cases) having a data file designated as recoverable has associated with it a recovery file. The recovery file is created by the system when the first before image for a recoverable file is about to be written.

All recovery files are created subordinate to your working directory, unless you specified otherwise by the Assign Recovery File (ARF) command. (The names of the files are recorded in the RECOVERY directory, which is positioned under the root directory of the system volume. This directory is maintained by the system.) Each recovery file is assigned a name of the form:

```
$$RECOV.ggtt
```

where gg is the group identifier and tt the task identifier.

File Recovery Process

The system recovers a data file (i.e., erases the updates made to it) by writing the before images back to the file.

You declare points in your processing (called cleanpoints) at which all file updates are considered valid. When a cleanpoint is declared, all before images taken up to that point are invalidated. New before images are written when you again begin to update the file.

You can perform a rollback at any time during processing. When a rollback is requested, the before images are written to the file, wiping out updates made since the last cleanpoint.

Use of the cleanpoint and rollback functions is recommended in a transaction-oriented environment.

TAKING CLEANPOINTS

When you consider the data in your file to be consistent and valid, declare a cleanpoint in your program. Cleanpoints are established by CALL "ZCLEAN" statements in COBOL programs or $CLPNT macro calls in Assembly language programs. When a cleanpoint is declared, the system performs the following actions:

- Writes all modified buffers to disk.
- Updates all directory records.
- Invalidates the recovery file before images that have been taken for the data file.
- Unlocks all records previously locked by the user. (Tasks waiting for these records are activated.)

The File System automatically performs a cleanpoint when a recoverable file is closed.

REQUESTING ROLLBACK

Rollback initiates the recovery of a file to the condition in which it was at the last cleanpoint. If programming in COBOL, request a rollback by coding a CALL "ZCROLL" statement. If programming in Assembly language, request a rollback by coding a $ROLBK macro call. When you request a rollback, the system performs the following actions:

- Takes before images from the recovery file and writes them to the data file, thereby wiping out updates made since the last cleanpoint.
- Invalidates the before images on the recovery file.
- Unlocks all records previously locked by the user. (Tasks waiting for these records are activated.)

The File System performs a rollback when a task group terminates abnormally.

RECOVERING AFTER SYSTEM FAILURE

If recovery files exist, the operator should issue the Recover command so that the system will perform a rollback of all recoverable data files. (Refer to the System User's Guide for details.)

FILE RESTORATION

File restoration provides the ability to preserve updates that have been made to files and to apply these updates to saved versions of the files if the original versions become corrupted. You cause images of records that have been modified (after images) to be recorded in a journal (after image) file. You can then use the journal file in conjunction with the Save, Restore, and Roll Forward commands to restore files to a known state if data in the files is destroyed as a result of a device failure (e.g., if I/O errors indicate any damaged files and/or volumes, file restoration procedures are recommended).

Designating Restorable Files

You designate files as restorable by specifying the -RESTORE argument of the CR command. Files not created as restorable can be made restorable by specifying the -RESTORE argument of the MFA command. (Restorable files can be made nonrestorable by specifying the -NORESTORE argument of the MFA command.)

It is recommended that files designated as restorable also be designated as recoverable (having the -RECOVER attribute) to provide for complete file integrity if a device or system failure occurs.

Journal File Creation

The journal file is created and maintained by the operator through the Open Journal, Close Journal, Display Journal, and Swap Journal commands. One system-wide journal file records updates made to all restorable disk files. The journal file can be a tape or disk file.

Each time a record in a restorable file is updated, the system records on the journal file the image of the record as it exists before modification (the before image) and after the modification (the after image). The after image of the updated record is written to the journal file at the time the record in the file is physically updated.

The journal file contains a running summary of all changes made to restorable files (e.g., if a restorable file is renamed or modified, appropriate entries are added to the journal file to reflect these changes). Restorable disk files cannot be modified in any way unless the journal file has been previously opened by the operator.

File Restoration Process

For each file that is corrupted, the restoration process involves mounting a known valid version of the file, reconstructed from data preserved during a previous save operation. The save operation involves preserving the data contents and selected attributes of the uncorrupted file (by means of the Save command) before any catastrophe occurs, then restoring the file structures of the saved file (using the Restore command) after the file has been corrupted. Following these actions, you cause after images from the journal file to be applied to the restored file (using the Roll Forward command). The restored file now incorporates the changes or updates stored in the journal file since you last invoked the Save command.

File restoration offers more extensive procedures if files are corrupted following a device failure and file recovery procedures fail to return files to a consistent state.

For example, the operator opens the journal file and enters the Recover command. If the Recover command executes successfully, you can log in and continue processing. If the Recover command fails to execute successfully, the operator must close the journal file, mount saved versions of all files, and enter the Restore command. The Roll Forward command is then specified. This command applies journal file images to all restored files, thereby updating the files to reflect modifications made after save commands were entered for those files. File restoration is then complete and users can log in and continue processing.

CHECKPOINT RESTART

The Checkpoint Restart facility allows you to provide a file recovery and program restart capability in a batch processing environment. Through Checkpoint Restart, you can establish a point in your program to which you can return at any time and continue processing. This return point (called a checkpoint) is used to save the current status of the task group request. You can perform a restart to the most recently completed checkpoint after the abnormal termination of the task group request or at any point during the processing of the task group request. A restart cannot be performed from an earlier checkpoint, nor can it be performed after the normal termination of a task group request.

Checkpoint Restart does not support the use of the Listener secondary login facility.

Checkpoint

When a task requests a checkpoint, the system records the current contents of user memory and the current state of tasks, files, and screen forms onto a checkpoint file the user has previously assigned. The system then takes a cleanpoint so that recoverable files are synchronized with that checkpoint. (Refer to "File Recovery" earlier in this section for a description of recoverable files and cleanpoints.)

The system supports one checkpoint task and any number of other tasks that are dormant or waiting on requests placed against other tasks in the task group. (Thus, a single active command executing under the Command Processor and/or any number of nested EC files can be checkpointed.)

CHECKPOINT FILE ASSIGNMENT

You enable the Checkpoint Restart facility for your task group and designate where its checkpoint images are to be recorded by issuing the Checkpoint File Assignment (CKPTFILE) command.

Checkpoints are written alternately to each of a pair of checkpoint files. This technique ensures the availability of the previous valid checkpoint if a failure occurs during the process of taking a checkpoint. The system locates and uses only the most recently completed successful checkpoint from the pair of checkpoint files that you specified.

When designating the checkpoint file, specify a single pathname (the last element of which can be a maximum of 10 characters). The system appends the suffixes .1 and .2 as appropriate. If the system cannot find one or both of the specified checkpoint files, it creates it (them).

TAKING A CHECKPOINT

When a checkpoint is taken, the system writes a checkpoint image and performs a cleanpoint for all recoverable files. If programming in Advanced COBOL, request a checkpoint by coding a CALL "ZXCKPT" statement or using the RERUN clause in the I-O-CONTROL paragraph. If programming in Assembly language, request a checkpoint by coding a $CKPT macro call.

Your task group must be in a checkpointable state when it requests a checkpoint. A task group is in a checkpointable state when each task that is part of the group has requested a checkpoint, is waiting on a request issued to another task in the task group, or is dormant (i.e., there are no current requests for the task).

Once a checkpoint is recorded by a task group, it remains available as a restart point until the next checkpoint request is completed, the current checkpoint file is disassigned (by the -DISASSIGN argument of the CKPTFILE command), or the task group request is terminated normally.

The lead task of the group may be waiting on both another task that is a member of the group and a "break" request.

CHECKPOINT PROCESSING

When a task group takes a valid checkpoint, the system records the following information on the checkpoint file established for that group:

1. Executive information including data structures, user pool memory blocks, data segments of bound units linked with separate code and data, and floatable overlays.

2. Status and pathnames of the standard I/O files and nonsharable bound units.

3. Memory locations and pathnames of sharable bound units.

4. Current state of screen forms for terminals operating under the display formatting and control facility.

5. Status and position of all active user files (i.e., files that have been associated, reserved, or opened).

When your file information has been recorded, the checkpoint image is completed and a cleanpoint is taken. You must ensure that files to be synchronized with the checkpoint restart process have been designated as recoverable. Since the File System performs a cleanpoint when a recoverable file is closed, you may have to take a checkpoint prior to closing the file to keep checkpoint restart synchronized with the state of the recoverable file. (Temporary files cannot be designated as recoverable.)

Checkpoints cannot be taken while an active local mail message group exists (i.e., a checkpoint cannot be taken in the period between message initiation or acceptance and message termination).

Checkpoints are not made automatically obsolete by the normal termination of the task under which they were issued. To invalidate a previous checkpoint (taken during the execution of one command) before processing a new command, you must take a checkpoint immediately prior to the termination of that command.

Restart

You can perform a restart at the following times:

- During the processing of the task group request that issued the checkpoint request.
- During the processing of a task group request that was scheduled after the abnormal termination of the task group request in which the checkpoint was taken.
- When the system is reinitialized following a system failure.

When a restart request is issued, the task group issuing the request is terminated abnormally and the task group request recorded on the checkpoint file is again put into effect.

The system locates the most recently completed checkpoint and reads the checkpoint image from the file, rebuilding the Executive data structures and memory blocks, reloading bound units, and repositioning active user files.

Procedural code and workspace must occupy the same physical memory locations that were used when the checkpoint was taken. In general, task groups that are to be restarted must be the sole users of exclusive memory pools. Sharable bound units referred to by these groups must be permanently loaded (through the Load command in the system startup EC file). The configuration under which the restart is performed must be identical to that which existed when the checkpoint was taken.

REQUESTING A RESTART

To restart from the last completed checkpoint (and to abort the current task group request if restarting during the session), issue the Restart command. The operator can restart an existing task group that has a valid checkpoint by using the -GROUP argument of the Restart command. If the memory blocks required to effect the restart are not available, the restart is aborted. Specification of the -WTMEM argument of the Restart command causes the system to wait until the specific memory blocks required to perform the restart become available.

If this is a restart following a system failure, the Recover command must have been issued by the operator or through an EC file to perform a system-wide rollback of all recoverable files.

If a restart is performed during a session, the abort (termination) of the group request causes a rollback of all recoverable files in your task group. The abnormal termination of the group request causes the last completed checkpoint image to be retained as a valid checkpoint. The Abort Group and Abort Group Request commands force an abnormal termination; the Bye command causes a normal termination. (The normal termination of the Command Processor with a nonzero value in the $R2 register is treated as an abnormal termination for checkpoint file purposes.)

The Validate Checkpoint command or active function can be used to ascertain whether the specified checkpoint file pair contains a valid restartable checkpoint.

RESTART PROCESSING

When you issue the Restart command, the system performs the following steps:

1. Locates the most recently completed checkpoint.

2. Validates that the restart is being performed under the same user id as that used when the checkpoint was taken.

3. Rebuilds Executive data structures.

4. Reads nonsharable bound units, data segments, floatable overlays, and memory blocks that were obtained by get-memory operations from the checkpoint image into the same memory locations they occupied at the time the checkpoint was taken.

5. Reloads sharable bound units in the system memory pool. Only the code segment is reloaded if the bound unit was linked with separate code and data. Unless it was linked with the restart relocatable attribute (Linker RR directive), the code segment is reloaded at the same system pool memory locations occupied when the checkpoint was taken.

6. Associates, gets, opens, and positions active user files recorded on the checkpoint image. Rollback should have been performed already (refer to the previous paragraph "Requesting a Restart").

7. Restores the screen content of terminals that were operating under the display formatting and control facility and were active at the time of the checkpoint.

8. Reissues the break request if such a request had been issued by the lead task at the time of the checkpoint.

9. Turns on the task that issued the checkpoint request at the next sequential instruction after the checkpoint.

The checkpointed state of the standard I/O files is reestablished at restart time. Modifications made to files (e.g., EC files) between the checkpoint and the restart must be restricted to those that do not invalidate the repositioning of the files. A command being restarted must remain in the same position in the file; only those commands that follow the restarted command have any effect on the restarted task group request.

Sharable bound units being used by a checkpointed task group are reloaded and not restored from a checkpointed memory image (except for the data segments of bound units linked with separate code and data). Thus, all such bound units should contain only code. All sharable bound units in use by a restarting task group must be identical to the versions that existed at the checkpoint. They cannot be relinked. If an Overlay Area Table (OAT) is in use for such a bound unit, no overlay area can be reserved at the time the checkpoint is taken.

If you have application programs that issue physical I/O orders for communication devices, you must reissue connects to those devices before issuing read and write orders to them.

GLOSSARY

Appendix A
GLOSSARY

HT (Horizontal Tab)

Command Processor: Reserved character.

Δ (space or blank)

Command Processor and Utilities: Reserved character; separates arguments and commands. Operator Interface Manager: At the beginning of a line, interrupts output.

! (exclamation point)

File System: A prefix indicating a physical device (sympd) name (e.g., !LPT00). Line Editor: Escape character (e.g., !F).

" (quotation mark)

Command Processor: Reserved character delimiting strings that contain embedded blanks (e.g., "D. COOK"). See ' (apostrophe).

# (number sign)

Line Editor: Signifies condition in If Data, If Range, and If Line directives. Linker: Specifies the current address.

$ (dollar sign)

Line Editor: In an address expression, represents the last line of the buffer (e.g., $P). In any other Line Editor expression, represents the end of a line (e.g., /DIVISION.$/). Linker: Specifies the next location (e.g., BASE $). File System: First character of a macro call name or mailbox (e.g., $GTFIL).

% (percent sign)

Linker: Address argument representing the location one word greater than the highest address previously used in a linked root or overlay (e.g., LDEF XTAG,%). Copy, Compare, Compare ASCII, and Rename Commands: Represents the character in the corresponding component and letter position of the entry name (e.g., START_U%.EC).

& (ampersand)

Line Editor: Used in the string expression of the Substitute directive to indicate that the current expression is to be repeated (e.g., S/TO BE/& OR NOT &/). Multi-User Debugger (numeric) and $D Debug: Address symbol, representing the next address beyond the address used in the previous debug directive. Command Processor: Reserved character. Indicates continuation of a command on more than one line. Execute Command: Indicates EC directives and comment lines (e.g., &P BEGIN LINK). TCL Compiler: Indicates continuation of a statement on more than one line.

' (apostrophe)

Command Processor: Reserved character. See " (quotation mark).

() (parentheses)

Command Processor: Delimits components of an iteration set (e.g., PRINT (FILEA FILEB)). Multi-User Debugger (Numeric) and $D Debug: Delimits action lines to be stored for later use. Line Editor: Delimits multicharacter buffer name; optionally, delimits single-character buffer name (e.g., B(EXEC)). TCL Compiler: Indicates insertion of field value.

* (asterisk)

Line Editor, CLM, TCL Compiler: Designates an expression. Patch: Comment directive. File System: Represents one component of a file name (e.g., COBPRG.*). In relation to Access Control Lists (ACLs) and Common Access Control Lists (CACLs), represents any user, account, and/or mode (e.g., COOK.*.INT). List Profile Utility, Multi-User Debugger (Numeric) and $D Debug: Signifies "all."

+ (plus sign)

Line Editor: Indicates unary addition of an address (e.g., +2P, 2+3). Multi-User Debugger (Numeric) and $D Debug: Performs addition.

, (comma)

Line Editor: Separates two addresses to be referenced inclusively (e.g., 1,5P). CLM, Linker, Sort, and Merge: Separates arguments within directives.

- (minus sign)

Command Processor: Immediately precedes an argument (e.g., -ECL). Line Editor: Indicates unary subtraction of an address (e.g., -2P). Multi-User Debugger (Numeric) and $D Debug: Performs subtraction.

. (period, decimal point)

File System: (1) Separates an entry name into components (e.g., COBPRG.C). (2) Used as a single element at the beginning of a pathname to indicate the working directory (e.g., .>FILE_DUMP). Line Editor: (1) In an address, represents the current line of the buffer (e.g., .P). (2) In an expression, requests a string containing any character (e.g., /PROG.AM/). Multi-User Debugger and $D Debug: Address symbol, representing the same starting address used in the previous debug directive. TCL Compiler: Indicates the end of a statement.

/ (slash)

File System: If first character of a star name, negates the meaning of the star name (e.g., /*.WORK). See * (asterisk). Line Editor: Delimits strings in Expressions and Substitute directive (e.g., S/OLD/NEW/). Patch and File Change: Immediately precedes a relative location or offset. Multi-User Debugger: Separates location from repetition value. $D Debug: Separates directive from the LRN of the output device and the location from the repetition value. Linker: Precedes a comment in a Linker directive file (e.g., /SECOND OVERLAY).

: (colon)

Line Editor: Indicates label definition (e.g., :7).

; (semicolon)

Line Editor: Separates two addresses; the first address becomes the current line, after which the value of the second address is calculated (e.g., 2;.3P). Patch: Separates arguments in Patch directives. Sort and Merge: Separates directives. Linker: Separates Linker directives on one line. Command Processor: Reserved character. Separates commands. Multi-User Debugger and SD Debug: Separates directives.

< (less-than)

File System: Indicates movement in the storage hierarchy toward the root and a change in one level in that direction (e.g., <LIBRARY). Assembler and Patch: Immediately precedes a relocatable address. Multi-User Debugger and SD Debug: Specifies the condition to be satisfied in an IF directive for conditional processing of the directive line.

= (equal)

Line Editor: Print Line Number directive. Multi-User Debugger (Numeric) and SD Debug: Expresses equality for an IF directive. Linker: Address argument, specifying the base address associated with the object unit identified by an associated label (e.g., BASE =OPNCRD). Copy, Compare, Compare ASCII, and Rename commands: Represents the corresponding component of a file name (e.g., COPY FILE.A =.B).

> (greater-than)

File System: (1) Used at the beginning of a pathname to indicate a file or directory under the User Root Directory (URD) (e.g., >SYSLIB2) and (2) Within a pathname, indicates movement in the storage hierarchy away from the root; connects two directory names or a directory name and a file name (e.g., ^MYVOL>MYDIR>MYFILE). Line Editor: Go To directive (e.g., >1). Multi-User Debugger and SD Debug: Specifies the condition to be satisfied in an IF directive for conditional processing of the directive line. Assembler and Patch: Indicates short displacement address.

>> (two consecutive greater-than signs)

File System: Used at the beginning of a pathname to indicate a file or directory under the System Root Directory (SRD) (e.g., >>SID).

? (question mark)

Line Editor: Address prefix directive. Copy, Compare, Compare ASCII, and Rename Commands: Represents any character appearing in the corresponding component and letter position of a file name (e.g., START_?P.EC). (See %.) File System and Command Processor: Immediately precedes a symbolic start address (entry point) in a bound unit name (e.g., NOW?TIME). In some commands, requests help (e.g., EP (Edit Profile)).

@ (at-sign)

Command Processor: Delete the previously typed character (e.g., TIMM@E).

[] (brackets)

Command Processor and TCL Compiler: Delimits active functions (e.g., (&P THE TIME IS [TIME]). Multi-User Debugger (Numeric) and SD Debug: Signifies the contents of the location defined by the expression within the brackets.

^ (circumflex)

File System: (1) Indicates a root directory, and must immediately precede a root directory name (e.g., ^SYSRES) and (2) Used as a single element at the beginning of a pathname to indicate the root of the working directory (e.g., ^>MYDIR). Line Editor: (1) When designated as the first character of a string, requests lines beginning with the string, excluding the circumflex (e.g., /^IDENTIFICATION/) and (2) Indicates negation in certain directives. Multi-User Debugger (Numeric) and SD Debug: Indicates negation as part of an IF directive.

_ (underscore)

File System: Joins two or more words in a file or directory name that the system is to interpret as one word (e.g., LIST_PROG).

| (vertical bar)

Command Processor: Suppresses rescanning for returned active strings.

abbrev, login

See login abbreviation

abbreviation, login

See login abbreviation.

abort

An operator action resulting in the immediate cessation of operation of a task group or the operation of the currently executing request in a task group. All resources are returned to the Executive. The bound unit of the lead task of an aborted request may be retained.

absentee

A processing mode characterized by the absence of interaction between the user and the system during execution of a user's program. The terms "absentee" and "batch" are used synonymously.

Access Control List (ACL)

A list specifying which user(s) can use the resource with which the list is associated.

ACL

See Access Control List.

activate

An operator action resulting in the resumption of a previously suspended task group. (See Suspend.)

Active

A task is in the Active state when it is executing or ready to execute, when its priority level becomes the highest active one in the central processor.

active function

A form of a command whose output string is placed in the command line before the rest of the line is processed.

active level

The priority level currently in effect.

address, absolute

A reference to a storage location that has a fixed displacement from absolute memory location zero.

address, relocatable

A reference to a storage location that has a fixed displacement from the program origin, but whose displacement from absolute memory location zero depends upon the loading address of the program. (See relocation factor.)

administrator, system

Person responsible for authorizing and registering users in the user profile, assigning passwords, and controlling the resources available to specific user profiles.

after image

The image of a record in a restorable disk file as it exists after alteration. Written to a system journal file.

algorithm

A set of well defined rules for the solution of a problem.

alternate index organization

Alternate indexes are used to view a file ordered with a different key. The same data file can be ordered in many different ways by having more than one alternate index.

application program

A user-written program for the solution of a business, industrial, or scientific problem.

area

A DM6 I-D-S/II integrated file.

argument

User-selected items of data that are passed to a procedure (e.g., system service macro call arguments that are passed to the called system service, or command arguments passed to the invoked task). Synonymous with arg. (See parameter.)

argument, control

A keyword whose value specifies a command option. (See keyword.)

argument, positional

An argument whose position in the command line indicates to which variable the item of data is applied.

ASCII (American Standard Code for Information Interchange)

The interchange code established as standard by the American Standards Association.

asynchronous

Without regular time relationships. As applied to program execution, unpredictable with respect to time or instruction sequence.

attribute, file

Any of a set of disk file characteristics established when the file is created or modified to include such integrity features as recovery, restoration, and record locking.

base level

(See priority level, base.)

batch

An execution environment used primarily for non-real-time activities such as program development. (See absentee.)

batch pool

The memory pool from which the batch task group is supplied memory segments.

batch task group

The single task group that executes in absentee mode. It owns a set of resources: the batch memory pool, and the peripheral devices currently available to it.

BCB

(See Buffer Control Block.)

BCD

Binary-Coded Decimal notation.

before image

A copy of a record from a recoverable disk file, as it exists just prior to updating, written to a system recovery file.

Binary Synchronous Communications (BSC)

A communications procedure, using a standardized set of control characters and control character sequences, for the synchronous transmission of binary-coded data.

block

The logical unit of transfer between main memory and a tape file. The size of a block may be variable depending on the number of records and whether they are fixed or variable in length.

bootstrap loader

(See loader, bootstrap.)

bootstrap routine

A routine, contained in a single record that is read into memory by a Read-Only Memory (ROM) bootstrap loader, which reads the operating system into memory. (See ROM bootstrap loader.)

bound unit

The output of one Linker execution that is placed in one file. A bound unit is an executable program consisting of a root segment and zero or more related overlay segments.

Bound Unit Descriptor (BUD) block

A system control structure containing information provided by the Linker to describe a bound unit.

break

A user action, initiated by pressing the break or interrupt key, that interrupts a running task so that commands can be entered. After the break, the interrupted task can be restarted or terminated.

breakpoint, bound unit

A point set in a debugging program where instructions are inserted to monitor the Executive loading process.

breakpoint, quick

A point in a program where a 02 instruction is inserted to monitor time-dependent tasks.

breakpoint, true

A point in a program where a 02 instruction is inserted to interrupt execution and activate a debugging program to monitor task execution.

broadcast

A message sent to all logged-on users through the Send Message Mailbox (SMM) command.

BSC

(See Binary Synchronous Communications.)

BUD

(See Bound Unit Descriptor (BUD) block).

Buffer Control Block (BCB)

A control structure, contained in the system pool area, which describes the characteristics of the buffer.

buffer, Input/Output (I/O)

A storage area used to compensate for the differences in the flow rates of data transmitted between peripheral devices and memory.

buffer pool

A collection of storage areas to which the File System assigns disk files when they are opened. Shared files are assigned to public pools in system memory. Exclusive files are assigned to private pools in task group memory (or to public pools if no private pools exist).

building, system

(See system building.)

bus

(See Megabus.)

byte

A sequence of eight consecutive binary digits operated upon as a unit.

CACL

(See Common Access Control List.)

calling sequence

A standard code sequence by which system services or external procedures are invoked.

CCP

(See Channel Control Program.)

channel

A path along which communications can be sent.

Channel Control Program (CCP)

A microcoded program that resides in the Multiline Communications Processor (MLCP); the CCP processes data characters, protocol headers, and framing characters.

checkpoint

A point in the user's program to which control can be returned and processing resumed following a task group abort. When the user takes a checkpoint, the system records the current contents of user memory and the current status of tasks, files, and screen forms on a checkpoint file. (See restart.)

checkpoint file

A user-named file on which the system records the current status of the task group request when a checkpoint is taken. Checkpoint files are created in pairs and checkpoints are written alternately to each file.

CI

(See Control Interval.)

cleanpoint

A point in the user's processing at which all file updates are considered to be valid. (See also rollback.)

CLM

(See Configuration Load Manager.)

clock frequency

The line frequency, in cycles per second, that is the basis (coupled with the scan cycle) for calculating the interval between real-time clock-generated interrupts.

Clock Manager

A monitor component that handles all requests to control tasks based on real-time considerations, and requests for the time of day and date in American Standard Code for Information Interchange (ASCII) format.

clock request block

A control structure supplied by a task to request a service from the Clock Manager.

clock scan cycle

The time in milliseconds between clock-generated interrupts.

clock timer block

The control structure used by the Clock Manager to control the clock-related processing of tasks.

code, object

The code produced by a compiler or the Assembler. The object code requires further processing by the Linker to produce a bound unit. (See also object unit.)

code, source

The code or language used by the programmer when the program was written. Code that must be processed by a compiler or the Assembler and the Linker before it can be executed.

cold restart

Restart after system failure.

command

An order that is processed by the Command Processor.

command-in

Any file or device from which commands to the Command Processor are read.

command language

The set of commands that can be issued by a user to control the execution of the user's online or batch task.

command level

The state of the Command Processor, when it is capable of accepting commands, optionally indicated by the display of the RDY (ready) message.

command line

A string of up to 127 ASCII characters in the form: command_name_1 [arg_1...arg_n];command_name_2 [arg_1...arg_k] ..., where command_name_i is the pathname(s) of the bound unit(s) that performs the command's function. (See argument for a description of "arg."; see &.)

Command Processor

A software component that interprets commands issued by the operator or a user, and invokes the required function.

Commercial Processor

A central processor that includes an enhanced instruction set providing native commercial mode instructions.

commercial simulator

A software component that executes a set of business-oriented instructions.

Common Access Control List (CACL)

A list specifying the access rights to all files or directories subordinate to the directory in which the list is established.

communications device

A device that transfers data over communications lines and is connected through the MLCP.

Compile Unit (CU)

A program unit, produced by a single execution of a compiler or the Assembler, that requires further processing by the Linker to produce a bound unit. (See object unit.)

concurrency

The read or write file access that the reserving task group intends for its tasks and the read or write file access that the reserving task group allows to other task groups.

configuration

The procedure that involves the use of configuration directives to define a system that corresponds to actual installation hardware.

Configuration Load Manager (CLM)

A system component that reads a file of user-supplied directives and causes the system to be configured according to the contents of the directives.

control argument

(See argument, control.)

control character

An ASCII character intepreted by a device (such as a VIP) as having a keyboard control function.

Control Interval (CI)

The unit of transfer between main memory and the storage medium (primarily disk devices) and is comparable to a "block" for tapes. The size is specified by the user and remains constant for a file. For disk files, the size of the CI must be a multiple of 256 bytes. A Unified File Access System (UFAS) file is composed of CIs that are numbered, starting at one. The CI also determines the buffer size.

CRB

(See Clock Request Block.)

CRT

Cathode Ray Tube. (See Visual Information Projection.)

CTB

(See Clock Timer Block.)

CU

(See Compile Unit.)

cumulative file

A third disk file utilized by Level 4 error logging. (See hold file and raw file.) Statistics contained in the raw file can be analyzed by examining the contents of the cumulative file. The cumulative file contains performance histories for each monitored device or for memory. The cumulative file must be created before issuing any error logging commands that reference it. When you issue a UPD_CUM command, records are removed from the raw file and added to the cumulative file. The user can print the statistics for each device, all devices, and/or memory using the PR_CUM command. After the information is printed, the user can delete the information in the cumulative file by issuing the DEL_CUM command.

Daemon

A system task group that manages queued print and punch requests.

Data Base Control Block (DBCB)

DM6 I-D-S/II working storage associated with a particular run unit containing record buffers, currencies, and other control information.

Data Base Control System (DBCS)

The DM6 I-D-S/II run-time package, which interprets Data Manipulation Language verbs, accesses the data base, and returns results to the user work area.

Data Description Language (DDL)

A nonprocedural language used to describe a data base (the schema description) or a portion of a data base (the subschema description).

data management

A File System component that handles the transfer of logical records.

DBCB

(See Data Base Control Block.)

DBCS

(See Data Base Control System.)

DDL

(See Data Description Language.)

device driver

A software component that controls all data transfers to or from a peripheral or communications device. (See line protocol handler.)

Device Media Control Language (DMCL)

A nonprocedural language that describes the physical characteristics of a DM6 I-D-S/II data base including CI size, area size, data base size, and CALC header frequency.

device-pac

The adapter between a Mass Storage Controller (MSC) or Multiple Device Controller (MDC) and peripheral device (e.g., printer, diskette drive).

direct access

The method for reading or writing a record in a file by supplying its key value.

direct address

The method for reading or writing a record in any Unified File Access System (UFAS) file by supplying its simple key (control interval and line number).

directive

A secondary level order read through the user-in file to a secondary processor (e.g., Line Editor, Linker, Patch, Debug, and CLM (configuration) directives.)

directory

A special file containing a description of other files and/or subordinate directories.

disk

A generic name for mass storage devices such as diskette, cartridge disk, and storage module.

display processing

A method for developing, displaying, maintaining, and utilizing terminal display forms.

DM6 I-D-S/II

DM 6 Integrated-Data-Store/II. A CODASYL-based data base management system. (See also integrated file.)

DMCL

(See Device Media Control Language.)

dope vector

A structure for passing data items not aligned on word boundaries between programs.

Dormant state

The state of a task when there is no current request for that task.

Dual-Line Communications Processor (DLCP)

A programmable interface between a central processor and communications device(s) consisting of two lines.

EBCDIC

Extended Binary-Coded Decimal Interchange Code.

EC file

A file containing commands and (optionally) directives. In interactive mode, an EC file typically contains frequently used command sequences. In absentee mode, an EC file must contain all commands, directives, and anticipated user responses to program messages that will be needed for a session.

Edit Profile utility

An interactive program that allows the system administrator to register new users and/or to delete, list, enhance, and change the profiles of registered users.

EFN

(See External File Name.)

entry point

A start address within the root segment of a bound unit. By default, an entry point is the beginning of a procedure; the user can specify alternate entry point by symbolic address when he/she invokes a bound unit.

equal name convention

A special pathname convention that can be used with certain commands to automatically construct the output pathname entry name when the input pathname entry name has been resolved.

error logging

The collecting of memory and/or hardware-related error statistics for selected peripheral devices.

error-out

The file or directive by which the system communicates error information to the user or operator; established when a group request is entered.

exclusive memory pool

A fixed-partition memory pool whose boundaries do not overlap those of other pools. All of the tasks executing in exclusive memory pools share a common virtual view consisting of the memory assigned to all exclusive and nonexclusive memory pools and system global memory.

extent

A group of contiguous allocated sectors on a disk.

External File Name (EFN)

The absolute pathname of any file within the system. It must start with the ^ or > character. It has the form ^vol_id>dir_1>...>dir_n>filename for files on logically dismountable volumes and the form (>)>dir_1>...>dir_n>filename for files on the system volume. Devices can be referred to by the sequence !sympd. (See sympd.)

external procedure

A routine that is assembled or compiled separately from the program that calls it.

FCB

(See File Control Block.)

FDB

(See File Description Block.)

FIB

(See File Information Block.)

field

A specific area of a record used for a particular category of data.

file

A named collection of one or more records.

File Control Block (FCB)

A File System data structure that controls a user's access to a file. An FCB is pointed to by an entry in the logical file table and, in turn, points to an FCB. There is one FCB per user Logical File Number (LFN) associated with a file.

File Description Block (FDB)

A File System data structure that describes a file or directory. An FDB is pointed to by an FCB for a particular file. There is one FDB per file or directory currently known (reserved) in the File System.

File Information Block (FIB)

A user-created data structure containing required information for file processing.

file management

A File System component that handles the creation, deletion, reservation, opening, and closing of files.

file name

A 1- to 12-character name assigned to a collection of related data records, or to a peripheral or communications device. For a file on disk, this name is assigned when the file is created. For devices, the name is assigned at system configuration. (See pathname.)

file organization

A method that establishes a relationship between a record and its location in a file. (See indexed, relative, random, dynamic disk, or sequential file organization.)

file recovery

Ability to bring an uncorrupted disk file to a consistent condition after a software malfunction or system failure.

file restoration

Ability to reconstruct a disk file that has been corrupted due to device fault.

file set

A number of tape volumes used to contain one or more files. There are four types of tape volumes:

1. Monofile Volume - Contains only one file

2. Multivolume File - Contains one file on two or more tape volumes

3. Multifile Volume - Contains more than one file on one volume

4. Multivolume Multifile - Contains more than one file with any file spanning more than one volume.

File System

System software consisting of file, data, and storage management components that handles Input/Output (I/O) functions of the supported I/O devices.

First-In/First-Out (FIFO)

An execution or scheduling algorithm in which the first item received is the first item processed.

fixed-length record

A record stored in a file in which all of the records are the same length.

floatable overlay

An overlay that can be loaded into any available memory location within a task group's memory pool.

full duplex

Simultaneous independent transmission of data in both directions.

full pathname

An absolute pathname which, when specified, begins with a circumflex (^) (e.g., the root directory.)

function

A procedure that returns a single value to its caller. (See subroutine.)

globally sharable bound unit

A bound unit containing reentrant code and linked with the Gshare directive. A globally sharable bound unit is loaded in the system pool and can be used by any task in the system.

group control block

A system structure describing attributes of a task group.

group_id

(See task group identification.)

GRTS

General Remote Terminal Supervisor.

half duplex

Transmission of data in one direction at a time.

High Memory Address (HMA)

The address of the highest physical memory location in the central processor.

HMA

See High Memory Address.

hold file

A file that contains a copy of the Level 2 or Level 4 error logging statistics that are stored in memory. The hold file can be retrieved after system shutdown or crash. The hold file is automatically created when the operator specifies a SET_ELOG command with a logging Level of 2 or 4. The system assigns the hold file the name EL_HOLD. The hold file is automatically updated at intervals as specified in the SET ELOG command. Error logging need not be active to print statistics in the hold file using the PR_HOLD command. The statistics maintained in the hold file represent only the most recent memory or peripheral device statistics.

home directory

The user's initial working directory after logging in.

hot restart

Restart during a session.

IMA

(See Immediate Memory Addressing.)

Immediate Memory Addressing (IMA)

A form of addressing a location in main memory by referencing the location directly, indirectly, or through direct or indirect indexing.

independent memory pool

A fixed partition memory pool. All tasks executing in a specific independent memory pool share a common virtual view consisting of all memory assigned to that pool and system global memory.

indexed file organization

A disk file whose records are organized to be accessed sequentially in key sequence or directly by key value.

indirect extent

The group of contiguous allocated disk sectors that holds the relative volume number that contains the succeeding set of extents.

Input/Output (I/O) device

A peripheral or communications device.

Input/Output Request Block (IORB)

A control structure used for communication between a program and an I/O driver outside of the File System.

integrated file

A data base disk file whose records are accessed directly or sequentially using CALC keys and key values.

interactive

A processing mode characterized by a dialog between the user and the system during execution of a user's program.

interactive task

A task, which, when invoked, is under real-time control of user-specified directives.

interrupt

The initiation, by hardware, of a routine intended to respond to an external (device-originated) or internal (software-originated) event that is either unrelated, or asynchronous with, the executing program.

Interrupt Save Area (ISA)

An area used to store the context of an interrupted task. There is one ISA for each task in memory.

interrupt vector

A pointer to a priority-level-specific memory area called an ISA. There is one vector for each priority level, each having a dedicated memory location.

Intersystem Link (ISL)

A hardware element interconnecting two buses, thereby permitting the same functions between two units on different buses as between two units on the same bus.

IORB

(See Input/Output Request Block.)

ISA

(See Interrupt Save Area.)

ISL

(See Intersystem Link.)

journal file

A system file that contains a running summary of all changes made to all disk files designated as restorable.

key

An identifier for a specific record within a disk file.

keyword

A fixed-form character string preceded by a hyphen (e.g., -ECL). It can stand alone (e.g., -WAIT) or can be followed by a value (e.g., -FROM n).

KSR

A Keyboard Send-Receive teleprinter.

KSR-like terminal

A KSR teleprinter, CRT keyboard, or Visual Information Projection (VIP) terminal, which supports the Teleprinter (TTY*) protocol and is connected to the MDC, MLCP, or DLCP.

language key

A two-ASCII-character identifier used as a file name suffix to provide multiple national language support. The system default language key is specified at CLM time with the system default message library pathname. If the default message library pathname is defaulted, the language key is EN (English). At a primary login, the user is given the system default language key unless otherwise specified in the user's profile login default arguments with the -LK argument or (for single user profiles) directly in the login line.

lead task

The controlling task of a task group. The lead task can invoke other tasks to perform functions on its behalf (i.e., system services).

LFN

(See Logical File Number.)

LFT

(See Logical File Table.)

line

A record stored in a Series 60-compatible file.

line number

The relative position of a logical record within a control interval (CI). Line numbers start at zero for each CI.

Line Protocol Handler (LPH)

A communications program that processes messages, interrupts, and timeouts; handles protocol acknowledgement and error recovery; initializes the channel control program.

link

A process by which the Linker program combines separately compiled object units to produce a bound unit. Also, a communications channel between two modems.

Linker

A utility program that links one or more object programs into a single machine language relocatable program.

Listener

A system control component that allows a user to access the system through a selected set of terminals by means of Login commands.

load unit

A discrete program unit that has been compiled or assembled and linked. It is in machine language and is directly executable by the Executive. See bound unit.

Loader

A system control software component that dynamically loads from disk the root and overlays of a bound unit.

Loader, bootstrap

A utility program, usually permanently resident in main memory, that enables other programs to load themselves.

Logical File Number (LFN)

An internal identifier that becomes associated with a file when it is reserved. LFNs are used in all file references until the file is removed.

Logical File Table (LFT)

A data structure for use by the File System. It contains an entry for each LFT.

Logical Resource Number (LRN)

An internal identifier used to refer to tasks or devices.

Logical Resource Table (LRT)

A data structure within a task group containing an entry for each LRN used in an application, or a data structure within a system task group containing an entry for each LRN representing a device. Each entry is a pointer to the Resource Control Table (RCT).

Login

A command entered at a terminal monitored by Listener that is used to gain access to the system. The Login command spawns a task group to be associated with the user's terminal for a primary login or passes the user to an existing task group for a secondary login.

login abbreviation

A one-character type-in that is defined in the terminals file as an abbreviation for a complete login line. A login abbreviation may apply only to a specific terminal or may be used at all terminals in the system.

login parameters, default

Login line parameters stored in a user's profile. When a user logs in, these parameters are combined with arguments from the terminals file and/or arguments entered manually at login time to form the actual login line.

LPH

(See Line Protocol Handler.)

LRN

(See Logical Resource Number.)

LRT

(See Logical Resource Table.)

mail

Data contained in a mailbox directory.

mailbox

A special directory and a file within that directory that may contain data to be communicated to another task group (user).

MBZ

Must Be Zero.

MDC

Multiple Device Controller for peripheral devices other than cartridge disk, storage module, and magnetic tape.

Megabus

A set of parallel conducting paths connecting various hardware units of a computer.

memory dump

The representation of the contents of memory.

Memory Management Unit (MMU)

A hardware feature that intercepts all addresses generated by the Central Processor Unit (CPU) (virtual addresses) and transforms them to real memory addresses via its mapping array.

Memory Manager

A system control software component that controls dynamic requests to obtain/return memory from/to a memory pool.

memory pool

A block of central processor memory from which a task group obtains memory as required for executable code, control structures, and I/O buffers. (See swap, online, or system pool.)

memory save and autorestart unit

A hardware feature that can preserve the memory image through a power failure lasting up to 2 hours.

message

A communication of text that is to be displayed immediately at the receiving user's terminal.

M4_SYSDEF program

An interactive CLM directive generation program.

MLCP

(See Multiline Communications Processor.)

MMU

(See Memory Management Unit.)

MSC

Mass Storage Controller for cartridge disks or storage modules.

MTC

Magnetic Tape Controller for magnetic tapes.

Multiline Communications Processor (MLCP)

A programmable interface between a central processor and one or more communications devices. Can be programmed to handle specific communications devices.

multiprogramming

An operating system capability that allows the concurrent execution of tasks from more than one task group.

multitasking

An operating system capability that allows the concurrent execution of more than one task in one or more task groups.

multivolume set

A number of disk volumes that contain one or more files. An online multivolume set allows data for a single file to be distributed over many volumes. It requires that all volumes be mounted and available for the file to be used. A serial multivolume set permits sequential files to extend onto other volumes. The volumes can be mounted one at a time and can be used for very large sequential files.

NATSAP

Next Available Trap Save Area Pointer.

nonexclusive memory pool

A fixed-partition memory pool whose boundaries can overlap those of other nonexclusive memory pools. All tasks executing in nonexclusive memory pools share a common virtual view consisting of the memory assigned to all exclusive and nonexclusive poools and system global memory.

nonfloatable overlay

An overlay that is loaded into the same memory location relative to the root each time that it is loaded.

non-time-shared

An operating mode in which the task is active for the entire interval of time it requires to execute (unless interupted by a task of higher priority). (See time-shared.)

OAT

(See Overlay Area Table.)

object unit

A relocatable program unit produced by a single execution of a language compiler, or by the Assembler, and requiring further processing by the Linker to produce a bound unit.

OIM

(See Operator Interface Manager.)

online

An execution environment intended for use by application programs, including those operating in real time.

online pool

A fixed-partition memory pool from which an online task group is supplied memory. An online pool can be shared by more than one task group. (See exclusive, nonexclusive, or independent memory pools.)

online task group

A task group that executes in the online dimension; its resources are an online memory pool and the peripheral devices it requests.

operating system area

The memory area containing operating system software, user-written extensions to the operating system, and device drivers.

operator

Person who starts up the system each day, controls processing, manages peripheral devices, monitors system states, and regulates absentee jobs.

operator commands

The set of commands that can be issued by the operator to control online and batch execution.

Operator Interface Manager (OIM)

A system control software component that manages all messages sent simultaneously by multiple task groups to the operator terminal or from the operator terminal to a task group.

operator-out

The file or device by which an interactive command communicates with the system operator; established at system initialization or when a File Out command is issued.

operator terminal

A Keyboard Send/Receive (KSR)-like terminal specified for use in interactive communications between the operator and vendor-supplied and user-written application programs.

overlay

A section of a program that can be loaded during execution to overlay another section of the program. Used when there is insufficient memory to accommodate all the code of a program. (See floatable overlay and nonfloatable overlay.)

overlay area

An area of specified size into which floatable overlays are loaded.

Overlay Area Table (OAT)

A data structure containing parameters that control the use of overlay areas.

pacing rate

The frequency at which each new output line appears on an output display.

parameter

The data received by a procedure that is written in a generalized form to handle any data passed to it. See argument.

password

A unique combination of characters, initially established at registration, that identifies a user. A system control component verifies the password before granting access to the system.

patch

A portion of code used to modify an existing object or load unit on disk or in memory.

pathname

A character string by which a file, directory, or device is known in the File System.

pathname, absolute

A pathname that begins with a greater-than sign (>) or a circumflex (^). In the former case, it is a partial pathname and is appended to the root directory name of the system volume to form a full pathname; in the latter case, it is a full pathname and is used without modification.

pathname, device

A pathname by which reference is made to a peripheral device. Device pathnames have the general form !device_id.

pathname, relative

A pathname that does not begin with a greater-than sign (>) or a circumflex (^). It is a partial pathname consisting of one or more directory names and/or a file name, and is appended to the working directory pathname to form a full pathname.

pathname, simple

A special form of a relative pathname consisting of a single directory name or file name. It is appended to the working directory name to form the full pathname.

peripheral device

A device connected through the MDC, MSC, or MTC (e.g., a card reader, disk, or tape).

Physical Input/Output (PIO)

Physical Input/Output, or physical I/O, that is initiated through a request I/O macro call, outside of the File System, using IORBs.

PIO

(See Physical Input/Output.)

pool identifier

A two-character name, established a system configuration, by which a memory pool is identified, and by which a task group is assigned a memory pool when the task group is created.

positional argument

(See argument, positional.)

power resumption

A system facility that controls the restarting of the execution environment following a power failure.

primary login

The form of login that requests Listener to spawn a task group that has the terminal from which the login originated as its primary system file (i.e., the terminal will be the initial user-in, command-in, error-out, and user-out files).

priority level

A numeric value that can be assigned to a task or device for purposes of controlling processing. Values range from 0 to 63. The lowest values (highest priorities) are reserved for system tasks; Level 63 is the system idle level. Intermediate levels are available for user assignment to tasks and devices. The physical level at which a task executes is the sum of the highest level number assigned to a configured device plus three, the base level of the task group, and the relative level of the task within the group.

priority level, base

The priority level, relative to the system priority level, at which all tasks in a task group execute. A base level of 0 is the next higher level above the last (highest) system priority level.

priority level, hardware

A numeric value from 0 through 63 that can be assigned to a task or device to control processing. The lowest values (highest priorities) are reserved for certain system tasks. Level 62 is reserved for user tasks. Level 63 is the system halt level.

priority level, physical

(See priority level.)

priority level, relative

The priority level, relative to the base level, at which a user task within a task group executes. Relative Level 0 is the base level.

priority level, system

The priority level assigned to system devices and tasks.

profile

(See report queue profile file or user profile.)

program name suffixes

A "point-letter" character string such as ".O" for object units or ".A" for Assembly language source units appended to a file name to identify it as a source, object, or list unit.

protected string

A character string containing reserved characters that is enclosed by protected string designators. (See reserved character and protected string designator.)

protected string designator

A pair of quotation marks or apostrophes that enclose a character string containing reserved characters. (See reserved character.)

PVE

Polled Visual Information Projection (VIP) Emulator.

quarantine unit

A unit of message text; the smallest amount of transmitted data that is available to the receiver.

random file organization

A disk file whose records are accessed directly or sequentially through CALC keys and key values.

range

The number of bytes transferred during an I/O operation.

raw file

A second disk file utilized by Level 4 error logging. The raw file is automatically created when the operator specifies a SET_ELOG command with a logging level of 4. The system assigns the raw file the name "EL_RAW". The raw file maintains a cumulative performance record for memory and/or each device being monitored. Information from residual records in the hold file is written to the raw file whenever error logging for a device is disabled or when error logging is reestablished at Level 4 after the system has shut down or crashed.

Memory error logging information is written to the raw file only if the operator specified a STOP_ELOG MEMORY command prior to the shutdown or crash. Since the raw file is a relative file whose size increases if records are not deleted, the operator should delete these records periodically. (See hold file.)

record

A user-created collection of logically related data fields. Records are treated as a unit by the user and can be fixed or variable in length.

record locking

A file access feature that controls contention for records within disk files shared by two or more task groups.

recoverable file

A disk file that has been identified as one that can be brought back to a previously established state in the event of a software malfunction or system failure. (See before image and file recovery.)

recovery file

A system-created file used to contain before images. (See before image.)

reentrant routine

A routine that does not alter itself during execution; a reentrant routine can be entered and reused at any time by any number of callers.

registration

Process,by which the system administrator introduces users and accounts into the system.

relative file organization

A file whose records are organized to be accessed sequentially or directly by their record position relative to the beginning of the file.

relative level

(See priority level, relative.)

relative record number

A number representing the position of a record relative to the beginning of the file. The initial record is relative record number 1.

report queue

A directory used to contain the pathnames of files queued for later transcription.

report queue profile file

A file that designates the characteristics of reports that will be entered in a report queue and printed or punched at a later time.

report spooling

The queuing and subsequent transcription of reports.

request block

(See IORB, Task Request Block (TRB), CRB, Semaphore Request Block (SRB).)

request I/O

The macro call, issued to a driver, that performs Physical I/O (PIO).

request queue

A threaded list of request blocks.

reserved character

An ASCII character to which special significance is attached. These characters are: space (blank), horizontal tab, quotation mark ("), apostrophe ('), semicolon (;), ampersand (&), vertical bar (|), left bracket ([), and right bracket (]).

resident bound unit

A bound unit that is permanently configured in memory as an extension to the operating system.

residual range

The difference between the number of bytes requested and the number of bytes transferred during an I/O operation.

restart

A user-initiated process in which the system locates the most recently completed checkpoint on the checkpoint file and reads the checkpoint image, rebuilding the Executive data structures and memory blocks, reloading bound units, and repositioning active user files. (See checkpoint.)

restorable file

A disk file that has been identified as one that can be reconstructed to its latest state following a device fault. (See after image and file restoration.)

return address

The address of the instruction in a program to which control is returned after a call to a subroutine. By convention, this address is usually stored in register B5.

RFU

Reserved for Future Use.

rollback

The process by which before images stored on a recovery file are written to a recoverable file, negating updates made since the last cleanpoint. This action restores the file to the state it was in when the cleanpoint was taken. Also see cleanpoint, before image, file recovery, and recoverable file.

ROM bootstrap loader

A firmware routine (activated by pushing the Load key on the control panel) that reads the first record from a designated disk into memory.

root directory

The primary directory on a mass storage volume; it is pointed to by the root directory pointer in the volume label. The name of the root directory is the same as the vol_id. MOD 400 supports a User Root Directory (URD) and a System Root Directory (SRD), which may reside on different volumes.

root segment

The controlling segment of a program. It is resident in memory during the entire execution of the program and can call overlay segments.

ROP

Receive-Only Printer.

RSU

Reserved for System Use.

Scientific Instruction Processor (SIP)

A hardware option on central processor models that executes a set of scientific instructions.

search rules

An ordered list of directories that are searched by the system when a bound unit is to be located and loaded or executed.

secondary login

The form of login that requests the Listener to transfer control of the user terminal to a specified task group. The specified task group must already exist and have an outstanding Request Terminal monitor call ($RQTML) for the secondary login to satisfy.

secondary user

A user whose login line contains a destination, the identification (usually by group-id) of a subsystem which has requested a secondary terminal. The user is attached to the subsystem until released by it.

sector

A 128-byte portion of a diskette track, or a 256-byte portion of a cartridge disk, cartridge module disk, or storage module track.

security

Limitation and control of the type of access a user has to directories, files, and the system itself.

semaphore

A software counter mechanism, available to Assembly language programs, and used to coordinate the use of task code or other resources such as files.

Semaphore Request Block (SRB)

A data structure used to control semaphore processing.

sequence number

The internal identification number assigned to a request in a task group request queue.

sequential access

The method of reading or writing a record in a file by requesting the next record in sequence.

sequential file organization

A file on disk or magnetic tape whose records are organized to be accessed in consecutive order.

sharable bound unit

A transient bound unit consisting of reentrant code linked with the share directive. A sharable bound unit is available for execution by any task assigned to the same memory pool.

sharable file

Any file that is usable by more than one task concurrently.

SIP

(See Scientific Instruction Processor.)

SIP Simulator

A software component that provides the same functionality as the SIP.

source unit

A program written in source language for processing by a compiler or an assembler. Source units are stored as variable sequential data files.

spanned record

A record that spans a control interval or block.

spawn

To create, request the execution of, and then delete a task or task group.

spooling

The technique for storing output on disk files for subsequent printing.

SRB

(See Semaphore Request Block.)

standard I/O files

The command-in, user-in, user-out, operator-out, and error-out files.

star name convention

A special pathname convention that can be used with certain commands to perform an operation on a group of files, thereby eliminating the need for separate commands for each file.

startup

The procedure that bootstraps a vendor-supplied, preconfigured system from disk to provide a minimum operating environment.

startup EC file

An EC file whose commands are executed at system startup or when a task group is activated.

states (task)

A task can be in the following states: Dormant, Active, Wait, and Suspend.

storage management

The File System component that handles the transfer of blocks and control intervals between main memory and secondary storage (e.g., disks, tapes, etc.).

subroutine

Any procedure that alters data in an area common to both the subroutine and its caller. Contrast with "function".

subsystem

A general purpose application-oriented facility that provides interactive users with their interactive capabilities and view of the system. A subsystem is generally identified directly with the lead task of a task group. A subsystem can either be primary-user oriented (supporting one interactive user per task group) or secondary-user oriented (supporting multiple interactive secondary users per task group).

subsystem switcher

A menu-oriented component of the User Productivity facility that allows a logged in user to switch from one subsystem to another without having to log out and back in again.

Suspend

An operator action resulting in the temporary cessation of execution of a task group; all resources are retained by the task group. (See activate.)

swap pool

A memory pool in which segmented memory management is used. Tasks assigned to a swap pool can be swapped to backing storage in order to make memory available to competing tasks. Each task executing in a swap pool has its own virtual view.

symbolic start address

Bound unit entry point.

sympd

A name assigned to each peripheral device when the system is built. The acronym sympd stands for "symbolic peripheral device."

system building

The process of specifying system variables, identifying the peripheral devices and (optional) communications environment to the system, and tailoring main memory to suit system and user needs.

system console

(See operator terminal.)

system directory

One of the directories that the system uses in its search for a bound unit to be loaded for execution.

system global memory

The memory of the fixed system area and the system. This memory is in the virtual view of all tasks, regardless of their specific memory pool assignments.

system pool

The memory area from which the system task group (GCB and TCB) and system global structures (e.g., BCB and FDB) are allocated, and the area where globally sharable bound units reside.

system service macro calls

Macro calls available to Assembly language programs to perform a wide variety of system control and File System service functions.

system task group

The task group in which all drivers, the clock, the Command Processor, and OIM execute.

task

A sequence of instructions that has a starting point, an ending point, and performs some identifiable function.

Task Control Block (TCB)

The system control structure that describes the task's characteristics, including the contents of the hardware Interrupt Save Area (ISA).

task group

A named set of one or more tasks with a common set of resources; the framework within which every user and system function operates.

task group identification

A two-character name by which a task group is known to the system.

task group resource

One of a set of elements associated with a task group that enables it to perform its function. A resource can be a task, a central processor priority level, central processor memory, or a peripheral or communications device.

Task Manager

A system component that handles task requests to activate, wait, or terminate tasks.

Task Request Block (TRB)

A data structure used by one task to request another task and communicate with it.

TCB

(See Task Control Block.)

terminal

An I/O device.

terminals file

A sequential file that names the terminals monitored by Listener, defines terminal-specific access constraints, and defines system-wide and terminal-specific abbreviations for login lines.

terminate

A system service macro call request issued by the currently executing task at the end of its normal processing.

terminated

A task state in which there is no current request for the task.

timeslicing

An optional feature that minimizes the ability of tasks that use large amounts of central processor time to interfere with interactive users.

transaction

An event that is entered, recorded, and processed by the system.

transaction processing

Online data processing in which individual transactions are entered from terminals, validated, and processed through all relevant procedures.

transient bound unit

A bound unit that resides in memory as long as there is a request for it.

transparent mode transmission

A data transmision mode that allows data consisting of bytes having any bit configuration to be transmitted over communications lines. Thus, control characters can be transmitted as data.

trap

A control transfer caused by an executing program. The transfer is made to a predefined location in response to an event that occurs during processing.

trap handler

A routine designed to take a particular action in response to a specific trap condition.

Trap Manager

A system control software component that handles an executing program's transfer of execution control to a predefined trap location.

Trap Save Area (TSA)

An area in memory in which certain information is stored when a trap occurs.

trap vector

A pointer to a trap handler. There is one vector for each possible trap condition, in dedicated memory locations.

TRB

(See Task Request Block.)

TSA

(See Trap Save Area.)

UFAS

(See Unified File Access System.)

Unified File Access System (UFAS)

A file organization developed to provide a predictable relationship between records and their location in the file. UFAS files are transportable across all levels of Series 60 software.

unit control character

(See control character.)

user

An entity that can make demands upon the system; can be a logged-in person, a system routine such as a Daemon, etc. A person logged in under two accounts is considered to be two users for system loading purposes.

user identification (user_id)

A field that identifies the current user of a task group.

user-in

The file or device from which a command function requiring directives (e.g., the Line Editor) reads its input; it is established when the group request is made. User programs can also read from this file.

user-out

The file or device by which an interactive command communicates with the user; established when a group request is made, or a File Out (FO) command is issued. User programs can also write to this file.

user profile

The user's registration information as maintained by the system administrator using the Edit Profile utility. The user profile establishes a login id and a unique password capability for each user, as well as other privileges and/or limitations granted to specific users.

user registration

A mode of MOD 400 operation that maintains a file of registered users which specifies their login defaults and individual access rights. For definitions of terms related to user registration, see the Glossary in the System Building and Administration manual.

variable-length record

A record stored in a file in which records have different lengths.

VIP

(See Visual Information Projection.)

virtual view

A virtual view consists of all of the memory pools to which a task executing within the view has access. A virtual view consists of one of the following combinations of memory pools:

- The system pool and none or more exclusive pools and none or more nonexclusive pools and none or one batch pool. This virtual view is also called a "regular, release 2.1 style pool set."
- The system pool and one independent (I-type) pool.
- The system pool and the swap pool.

Visual Information Projection (VIP)

VIP devices consist of a screen (CRT) and keyboard. Hard-copy receive-only printers can be added to some models.

vol_id

(See volume identifier.)

volume

A fixed or removable storage unit (e.g., storage modules, diskettes, cartridges, tapes) that may contain one or more files.

volume header

A unique record at the beginning of every disk or magnetic tape volume that carries information about the volume.

volume identifier (vol_id)

The unique name for a disk or magnetic tape volume that is contained in the volume header.

volume name

(See root directory.)

volume set

A number of disk volumes that contain one or more files. Online volume sets require that all volumes are mounted and are available for use. Serial volume sets can be mounted one volume at a time.

Wait

A task is in the Wait state when it causes its own execution to be interrupted until a time request is satisfied, until another task releases a semaphore, until another task terminates, or until an I/O operation terminates.

word

A sequence of 16 consecutive binary digits operated upon as a unit; two consecutive bytes.

working directory

A disk directory pathname associated with a task group. It begins with a root directory name and contains zero or more intermediate directory names. It is used by the File System software to construct a full pathname whenever a task group refers to a relative or simple pathname.

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: DPS 6 & LEVEL 6
GCOS 6 MOD 400 SYSTEM CONCEPTS

ORDER NO. CZ03-00

DATED December 1982

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. □

FROM: NAME ______________________________ DATE ______________

TITLE ______________________________

COMPANY ______________________________

ADDRESS ______________________________

____

CUT ALONG LINE

PLEASE FOLD AND TAPE –
NOTE U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	DPS 6 & LEVEL 6 GCOS 6 MOD 400 SYSTEM CONCEPTS	ORDER NO.	CZ03-00
		DATED	December 1982

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. □

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____