HONEYWELL

DPS 6 & LEVEL 6 GCOS 6 MOD 400 SYSTEM PROGRAMMER'S GUIDE — VOLUME I

SOFTWARE

DPS 6 & LEVEL 6
GCOS 6 MOD 400
SYSTEM PROGRAMMER'S
GUIDE - VOLUME I

SUBJECT

Description of System Software, Including Executive Routines, Drivers, and Line Protocol Handlers, Accessible to Applications Written in Assembly Language

SPECIAL INSTRUCTIONS

Sections of this manual dealing with communications supersede the Communications Processing manual, CB03-03.

SOFTWARE SUPPORTED

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

ORDER NUMBER

CZ05-00

December 1982

Honeywell

PREFACE

This manual provides information useful to the Assembly language programmer for designing, executing, and checking out applications.

The manual describes system services available to the programmer for:

- System control
- Input/output to peripheral devices
- Input/output to communications devices.

The system services described include:

- Executive routines that can be invoked by monitor calls or macro calls
- Drivers servicing peripheral devices
- Line protocol handlers servicing communications devices.

Macro calls mentioned in this volume are described more fully in the System Programmer's Guide, Vol. II (CZ06-00). Assembly language is described in the Assembly Language Reference (CZ38-00).

This manual covers the following topics related to program preparation, execution, and checkout:

- Gaining access to the system
- Naming and manipulating files
- Preparing a source program with the Line Editor
- Linking
- Debugging
- Taking memory dumps.

File No.. 1R13, 1S13

CZ05-00

Notational Symbols

The following symbols are used in this manual to define the format of command and directive lines:

Square brackets [ ] indicate an optional entry.

Braces { } enclose entries 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.

The character Δ indicates one blank space.

User Typeins

Shading ▮▮▮▮▮▮ Indicates user input to the system.

Heading Hierarchy

Each section and appendix of this document is structured acccording 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 (Lowest)	Initial Capital Letters, Not Underlined

USER COMMENTS FORMS are included at the back of this manual. These forms are to be used to record any corrections, changes, or additions that will make this manual more useful.

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

CONTENTS

CONTENTS

CONTENTS

CONTENTS

ILLUSTRATIONS

TABLES

TABLES

1. Introduction

Section 1
INTRODUCTION

Volume I of the System Programmer's Guide provides general information useful to the Assembly language programmer for designing, executing, and checking out applications. The following subsections describe more specifically the content and organization of the manual.

SYSTEM FUNCTIONS

Sections 2 through 12 of the manual describe services provided by the system that can be invoked or controlled by Assembly language programs.

System Service Macro Calls

Sections 2 through 4 describe system services (functions) that can be invoked by macro calls or monitor calls. These are services for system control, file management, record management, and input/output to peripheral and communications devices. Table 1-1 lists alphabetically the macro calls by which system functions can be invoked. Throughout this manual, functions are referred to by their corresponding macro calls.

The user can also invoke a function by a monitor call (MCL) instruction followed by the function's code. The function code assigned to each function/macro call is shown in column 3 of Table 1-1.

The manual provides an overview of functions belonging to the same group. In Section 2, for example, all the functions related to semaphores are listed together. Semaphores are there defined as a mechanism for the sharing of a resource among members of the same task group. The part played in this mechanism by each of the listed functions is briefly indicated. Thus, the manual informs the user of available macro calls and indicates their functional relationship.

Volume II of the System Programmer's Guide, by contrast, describes each macro call individually. The individual descriptions provide information (relating to macro call arguments and register contents) that enables the user to actually employ the call in the application.

Device Drivers and Line Protocol Handlers

Section 6 describes the system software used for transmitting data between applications and peripheral (non-communications) devices. The section deals mainly with the data structures and codes by which the user instructs the device drivers and by which the drivers report the status of requested operations. (Macro calls related to input/output are discussed in earlier sections.)

Section 7 provides an overview of line protocol handlers, which are used for transmitting data between applications and communications devices. Sections 8 through 12 describe in detail the ATD, STD, PVE, BSC, and TTY line protocol handlers.

PROGRAM PREPARATION AND CHECKOUT

The sections mentioned above describe system services that an Assembly language application can utilize. The remaining sections of the manual describe procedures for preparing, executing, and checking out an application, once it is designed.

Section 13 tells the user how to gain access to the system from a terminal. Section 14, "File System Conventions", explains conventions for naming and procedures for manipulating both files and directories. Section 15 explains how to create a source file using the Line Editor.

Programming considerations, such as trap handling and calling external procedures, are treated in appendices.

Information on assembling a program is provided in Appendix C. Section 16 explains how to link object modules. Debugging, Memory Dumps, and Patch facilities are described in Sections 17 through 20. Thus, the sequence of the last eight sections (Sections 13 through 20) roughly follows the sequence of procedures involved in program preparation and checkout.

Table 1-1. System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$ABGRP	Abort group	0D/0A	Task group control
$ABGRQ	Abort group request	0D/07	Task group control
$ACTID	Account identification	14/02	Identification and information
$ACTVG	Activate group	0D/09	Task group control
$ALARM	Alarm	15/25	Terminal operator communications
$ASFIL	Associate file	10/10	File management
$BUAT	Bound unit, attach	0C/09	Task control
$BUDT	Bound unit, detach	0C/0B	Task control
$BUID	Bound unit identification	14/06	Identification and information
$BULD	Bound unit, load	0C/0A	Task control
$BUXFR	Bound unit transfer	0C/07	Task control
$CANRQ	Cancel request	0C/01	Task control
$CIN	Command in	08/02	Standard system file I/O
$CKPFL	Checkpoint file	0D/11	Task group control
$CKPT	Checkpoint	0D/0F	Task group control
$CLFIL	Close file	10/55-10/57	File management
$CLPNT	Clean point	0C/13	File management
$CLRSW	Clear external switches	0B/02	External switch

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$CMDLN	Command line process	0C/08	Task control
$CMSUP	Console message suppression	09/02,09/03	Operator interface
$CNCRQ	Cancel clock request	05/01	Clock
$CNSRQ	Cancel semaphore request	06/01	Semaphore handling
$CRB	Clock request block	-	Data structure generation
$CRBD	Clock request block offsets	-	Data structure generation
$CRDIR	Create directory	10/A0	File management
$CRFIL	Create file	10/30	File management
$CRGRP	Create group	0D/03	Task group control
$CRKDB	Create file key descriptor block offsets	-	Data structure generation
$CROAT	Create overlay area table	07/0A	Overlay handling
$CRPSB	Create file parameter structure block offsets	-	Data structure generation
$CRRDB	Create file record descriptor block offsets	-	Data structure generation
$CRSEG	Create segment	0C/0C	Task control
$CRTSK	Create task	0C/02,0C/03	Task control
$CWDIR	Change working directory	10/B0	File management
$DFCKP	Defer checkpoint	0C/19	Task control

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$DFRHD	Defer request on head	01/0D	Request and Return
$DFRTL	Defer request on tail	01/0C	Request and Return
$DFSM	Define semaphore	06/04	Semaphore handling
$DIPSB	Device information parameter structure block offsets	-	Data structure generation
$DLDIR	Delete directory	10/A5	File management
$DLFIL	Delete file	10/35	File management
$DLGRP	Delete group	0D/04	Task group control
$DLOAT	Delete overlay area table	07/0D	Overlay handling
$DLREC	Delete record	11/30,11/31	Data management
$DLSEG	Delete segment	0C/0D	Task control
$DLSM	Delete semaphore	06/07	Semaphore handling
$DLTSK	Delete task	0C/04	Task control
$DQPST	Dequeue and post	01/0B	Request and Return
$DSFIL	Dissociate file	10/15	File management
$DSTRP	Disable user trap	0A/02	Trap handling
$ELEND	Error logging end	02/09	Physical I/O
$ELEX	Error logging information, exchange	02/07	Physical I/O
$ELGT	Error logging information, get	02/08	Physical I/O
$ELOG	Error logging table	-	Data structure generation
$ELST	Error logging, start	02/05	Physical I/O

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$ENTID	Entry point identification	14/07	Identification and information
$ENTRP	Enable user trap	0A/01	Trap handling
$EROUT	Error out	08/03	Standard system file I/O
$EXTDT	External date/time, convert to	05/04	Date/time
$EXTIM	External time, convert to	05/05	Date/time
$FIB	File information block	-	Data structure generation
$FIBDM	File information block offsets (data management access)	-	Data structure generation
$FIBSM	File information block offsets (storage management access)	-	Data structure generation
$GAFIL	Get file access rights	10/7C	File management
$GAPSB	Get file access rights parameter structure block offsets	-	Data structure generation
$GDTM	Get date/time	05/06	Date/time
$GIDEV	Get device information	10/66	File management
$GIFAB	Get file information, file attribute block offsets	-	Data structure generation
$GIFIL	Get file information	10/60	File management

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$GIKDB	Get file information, key descriptor block offsets	-	Data structure generation
$GIPSB	Get file information, parameter structure block offsets	-	Data structure generation
$GIRDB	Get file record descriptor block offsets	-	Data structure generation
$GMEM	Get memory/get available memory	04/02,04/03	Memory allocation
$GNFIL	Get name	10/3C	File management
$GNPSB	Get names parameter structure block offsets	-	Data structure generation
$GRFIL	Grow file	10/38	File management
$GRPID	Group identification	14/08	Identification and information
$GRPSB	Grow file parameter structure block offsets	-	Data structure generation
$GTACT	Get file accounting information	10/42	File management
$GTFIL	Get file	10/20	File management
$GTPSB	Get file parameter structure block offsets	-	Data structure generation
$GWDIR	Get working directory	10/C0	File management
$HDIR	Home directory	14/0B	Identification and information
$INDTM	Internal date/time, convert to	05/07	Date/time

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$IORB	Input/output request block	-	Data structure generation
$IORBD	Input/output request block offsets	-	Data structure generation
$KILLT	Kill (abort) task	0C/11	Task control
$MACPT	Message group, accept	15/01	Intergroup message facility
$MCME	Message group, cancel enclosure	15/06	Intergroup message facility
$MCMG	Message group, count	15/07	Intergroup message facility
$MDFIL	Modify file	10/41	File management
$MDPSB	Modify file parameter structure block offsets	-	Data structure generation
$MGCRB	Message group control request block	-	Data structure generation
$MGCRT	Message group control request block offsets	-	Data structure generation
$MGIRB	Message group initialization request block	-	Data structure generation
$MGIRT	Message group initialization request block offsets	-	Data structure generation
$MGRRB	Message group, recovery request block	-	Data structure generation
$MGRRT	Message group recovery request block offsets	-	Data structure generation

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$MINIT	Message group, initiate	15/02	Intergroup message facility
$MODID	Mode identification	14/03	Identification and information
$MRECV	Message group, receive	15/03	Intergroup message facility
$MSEND	Message group, send	15/05	Intergroup message facility
$MTMG	Message group, terminate	15/04	Intergroup message facility
$NCIN	New command in	08/06	Standard system file I/O
$NMLF	New message library	08/08	Standard system file I/O
$NPROC	New process	0D/0B	Task group control
$NUIN	New user input	08/04	Standard system file I/O
$NUOUT	New user output	08/05	Standard system file I/O
$OPFIL	Open file	10/50,10/51	File management
$OPMSG	Operator information message	09/00	Operator interface
$OPRSP	Operator response message	09/01	Operator interface
$OVEXC	Overlay, execute	07/00	Overlay handling
$OVLD	Overlay, load	07/01	Overlay handling
$OVRCL	Overlay release, wait, and recall	07/07	Overlay handling
$OVRLS	Overlay area, release	07/06	Overlay handling

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$OVRSV	Overlay area reserve, and execute overlay	07/05	Overlay handling
$OVST	Overlay status	07/03	Overlay handling
$OVUN	Overlay, unload	07/0C	Overlay handling
$PERID	Person identification	14/01	Identification and information
$PPNTL	Postpone request on tail	01/0E	Request and Return
$PRBLK	Parameter block	-	Data structure generation
$PRFAU	Profile record, accounting update	24/42	User registration
$PRFCR	Profile record, create	24/20	User registration
$PRFDL	Profile record, delete	24/30	User registration
$PRFGT	Profile record, get	24/10	User registration
$PRFIF	Profile record, get user information	24/12	User registration
$PRFUP	Profile record, update	24/40	User registration
$RBADD	Return request block address	01/07	Request and return
$RBD	Request block displacements	-	Data structure generation
$RBOOT	Reboot	20/06	Software reboot
$RBPRM	Modify reboot parameters	20/05	Software reboot
$RCLHD	Recall from head	01/0F	Request and return
$RDBLK	Read block	12/00-12/04	Storage management

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$RDREC	Read record	11/10-11/16, 11/19	Data management
$RDSW	Read external switches	0B/00	External switch
$RETRN	Return	-	Request and return
$RLDMP	Unlock dumpfile	20/04	Software reboot
$RLSM	Release semaphore	06/03	Semaphore handling
$RLTML	Release terminal	17/04	Terminal function
$RMEM	Return memory/return partial block of memory	04/04,04/05	Memory allocation
$RMFIL	Remove file	10/25	File management
$RNFIL	Rename file/rename directory	10/40	File management
$ROLBK	Roll back (recover) files	0C/14	File management
$RPDFC	Report message, display formatting and control	0F/04	Message reporter
$RPMSG	Report message	0F/03	Message reporter
$RQBAT	Request batch	0E/00	Batch
$RQCL	Request clock	05/00	Clock
$RQGRP	Request group	0D/00	Task group control
$RQIO	Request I/O	02/00	Physical I/O
$RQSM	Request semaphore	06/00	Semaphore handling
$RQSPT	Request specific terminal	17/02	Terminal function
$RQTML	Request terminal	17/03	Terminal function

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$RQTSK	Request task	0C/00	Task control
$RS	Restart	0D/10	Task control
$RSVSM	Reserve semaphore	06/02	Semaphore handling
$RWREC	Rewrite record	11/40,11/41	Data management
$RVFPW	Reverify password	24/01	User registration
$SDL	Set dial	1B/00	Communications
$SETSW	Set external switches	0B/01	External switch
$SGRPA	Set group attributes	0D/13	Task group control
$SGTRP	Signal trap	0A/03	Trap handling
$SHFIL	Shrink file	10/37	File management
$SHPSB	Shrink file parameter structure block offsets	-	Data structure generation
$SPGRP	Spawn group	0D/05	Task group control
$SPTSK	Spawn task	0C/05,0C/06, 0C/15	Task control
$SRB	Semaphore request block	-	Data structure generation
$SRBD	Semaphore request block offsets	-	Data structure generation
$STMP	Status memory pool	04/06	Memory allocation
$STTY	Set terminal file characteristics	10/45	File management
$SUSPG	Suspend group	0D/08	Task group control

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$SUSPN	Suspend for interval; suspend until time	05/02,05/03	Clock
$SWFIL	Swap file	10/5A	File management
$SYSAT	System attribute information, get	14/11	Identification and information
$SYSID	System identification	14/04	Identification and information
$TEST	Test completion status	01/02	Request and return
$TFIB	File information block offsets (data and storage management access)	-	Data structure generation
$TGIN	Task group input	14/0C	Identification and information
$TIFIL	Test file for input	10/62	File management
$TOFIL	Test file for output	10/63	File management
$TRB	Task request block	-	Data structure generation
$TRBD	Task request block offsets	-	Data structure generation
$TRMRQ	Terminate request	01/03,01/04	Request and return
$TRPHD	Trap handler connect	0A/00	Trap handling
$USIN	User input	08/00	Standard system file I/O
$USOUT	User output	08/01	Standard system file I/O

Table 1-1 (cont). System Service Macro Calls

Macro Call Name (1)	Function Description (2)	Function Code (3)	Function Group (4)
$USRID	User identification	14/00	Identification and information
$VLCKP	Validate checkpoint	0D/12	Task group control
$WAIT	Wait	01/00	Request and return
$WAITA	Wait any	01/01	Request and return
$WAITL	Wait on request list	01/01	Request and return
$WAITM	Wait on multiple requests	01/01	Request and return
$WIFIL	Wait file (input)	10/64	File management
$WLIST	Wait list generate	-	Data structure generation
$WLSTM	Wait list, generate multiple	-	Data structure generation
$WOFIL	Wait file (output)	10/65	File management
$WRBLK	Write block	12/10,12/11	Storage management
$WRREC	Write record	11/20-11/26	Data management
$WTBLK	Wait block	12/20	Storage management
$XFERU	Transfer user	17/06	Terminal function
$XPATH	Expand pathname	10/D0	File management
$XRETU	Transfer and return user	17/07	Terminal function

Section 2
SYSTEM CONTROL FUNCTIONS

This section summarizes and briefly describes the system control macro calls that provide user access to system control functions. The macro calls are presented according to their functional groupings (see Table 1-1, column 4) as follows:

Batch
Clock
Communications
Date/time
Error handling
Identification and information
Memory allocation
Message facility (intergroup)
Message Reporting
Operator interface
Overlay handling
Physical I/O
Request and return
Secondary user terminal
Semaphore handling
File system I/O
Task control
Task group control
Trap handling
User registration
Software reboot

See Volume II of this manual for a detailed description of each macro routine/call.

BATCH FUNCTIONS

The macro routine call for batch functions allows a mode of program execution that requires no personal interaction with the system. To use the batch functions, the user prepares a file that is to act as the command input file and the user input file. All commands and program input are read from this file by the batch task group when your request executes.

The macro routine/call is:

Request Batch Execution $RQBAT

CLOCK FUNCTIONS

The macro calls for clock functions allow user control of task execution according to an elapsed time period. These macro calls use the clock manager. The clock manager is a system component whose primary function is satisfying/completing task requests at a specified time or after a specified interval.

The clock manager services interrupts from the real-time clock. At each interrupt, the clock manager ascertains whether the time interval associated with a request to initiate execution of the task has been satisfied. Depending on information contained in the clock request block (see Appendix D), the system will do one of the following:

- Activate a task
- Schedule an indicated request block
- Release a semaphore.

The clock macro calls act to:

- Connect a clock request block to the timer queue
- Disconnect a clock request block from the timer queue
- Suspend the issuing task until an interval of time has passed
- Suspend the issuing task until a given date/time.

The clock function macro calls are:

- Cancel Clock Request $CNCRQ
- Request Clock $RQCL
- Suspend for Interval $SUSPN
- Suspend Until Time $SUSPN

Volume II describes the Clock Request Block ($CRB) macro call, which generates a clock request block.

COMMUNICATIONS FUNCTIONS

The macro call for communications functions allows the user to set a telephone number to be used for automatic dialing. The macro routine/call is:

Set Dial $SDL

Section 4 discusses macro calls, other than Set Dial, applicable to communications processing.

DATE/TIME FUNCTIONS

The macro calls for date/time functions allow the user to:

- Obtain the current internal date/time value
- Convert the internal date/time value to external date/time format
- Convert the internal date/time value to external time format
- Convert an external date/time value to internal format.

The date/time macro calls are:

- External Date/Time, Convert to $EXTDT
- External Time, Convert to $EXTIM
- Get Date/Time $GDTM
- Internal Date/Time, Convert to $INDTM

MESSAGE REPORTING

The macro calls for message reporting allow an application to display error or help messages at the user's terminal.

The macro calls specify the code of a message that the Message Reporter then retrieves from a message library.

The message reporting macro calls allow an application to:

- Display chained messages (i.e., after viewing the first message in the chain, the user can request further information)
- Substitute arguments for parameters in the message text (e.g., specify a device name in a "device disabled message")
- Return messages to an application buffer rather than to a terminal

- Display messages at terminals running in any of the following modes:

  - Command
  - Menu
  - Display formatting and control.

The message reporting macro calls are:

- Report Message ($RPMSG)
- Report Message, Display Formatting and Control ($RPDFC).

EXTERNAL SWITCH FUNCTIONS

A task group can control its own execution by using external switch function macro calls to modify its external switches. An external switch operates much like a hardware switch on an operator's control panel. External switches can be set and cleared with the Modify Switches (MSW) command or with the $SETSW and $CLRSW macro calls.

An external switch word is associated with each task group. Each bit in the word corresponds to an external switch. Thus, each task group can manipulate 16 switches. A user program can contain instructions or statements to determine the settings of one or more of these switches. The program can then set or clear these settings to control its execution logic.

The macro calls allow the issuing task to:

- Set switches
- Clear switches
- Read the current values of the switches.

The macro calls are:

- Clear External Switches $CLRSW
- Read External Switches $RDSW
- Set External Switches $SETSW

IDENTIFICATION AND INFORMATION FUNCTIONS

The macro calls for identification and information make available to the user the following information concerning the current task or task group:

Information	Macro Call
Home directory pathname	$HDIR
Bound unit identification	$BUID
System identification	$SYSID
Task group account identification	$ACTID
Task group input file name	$TGIN
Task group mode identification	$MODID

Information	Macro Call
Task group person identification	$PERID
Task group user identification	$USRID
Entry point identification	$ENTID
Group identification	$GRPID
System attribute information	$SYSAT

MEMORY ALLOCATION FUNCTIONS

The macro calls for memory allocation functions allow the user to dynamically obtain memory from the task group's memory pool, to return this memory when it is no longer needed, and to ascertain the amount of memory available in a specified pool.

The macro call that allocates a memory block has two forms: one form obtains a memory block of the specified size only; the other obtains the largest existing contiguous memory block if a block of the specified size cannot be found. The macro call that returns a memory block also has two forms: one form returns an entire memory block; the other returns a specified part of the block.

The macro calls are:

- Get Memory/Get Available Memory $GMEM
- Return Memory/Return Partial Block of Memory $RMEM
- Status Memory Pool $STMP

MESSAGE FACILITY FUNCTIONS

The message facility allows the task groups to exchange messages through a message queue called a mailbox. Before messages can be transmitted, the mailbox must have been created by means of the Create Mailbox command. Mailboxes are described in detail in the System User's Guide.

A message text consists of several nested units, or enclosures. The smallest unit is a record; the next largest unit, made up of records, is a quarantine unit; the largest, made up of quarantine units, is a message. A quarantine unit is the smallest amount of transmitted data that is available to the receiver. Because a message can comprise a group of records, it is called a "message group."

The transfer of messages is facilitated by three request blocks: message group control request block (MGCRB), message group initialization request block (MGIRB), and message group recovery request block (MGRRB). These data structures are tabulated in Appendix D and described in Volume II.

Message facility macro calls perform the following:

- Initialize communications between groups by setting values of the message group initialization request block (MGIRB)
- Validate the acceptor's access to an existing mailbox
- Ascertain the number of messages in a mailbox
- Identify the specific message to be accepted
- Request the receipt of a message, specifying values for the message group control request block (MGCRB)
- Delete the last record in an incomplete quarantine unit or delete the quarantine unit itself
- Send a message group
- Terminate a message group, normally or abnormally.

The message facility macro calls are:

• Message Group, Initiate	$MINIT
• Message Group, Accept	$MACPT
• Message Group, Count	$MCMG
• Message Group, Receive	$MRECV
• Message Group, Cancel Enclosure	$MCME
• Message Group, Send	$MSEND
• Message Group, Terminate	$MTMG

OPERATOR INTERFACE FUNCTIONS

The macro calls for operator interface functions enable tasks to communicate with the operator terminal by:

- Displaying a message on the operator terminal
- Sending a message to the operator terminal and receiving a response
- Activating or deactivating console suppression; i.e., suspending or restoring issuance of messages to the operator terminal for the issuing task group.

The macro calls are:

• Console Message Suppression	$CMSUP
• Operator Information Message	$OPMSG
• Operator Response Message	$OPRSP

The $OPMSG and $OPRSP macro calls require input/output request blocks (IORBs), which can be generated by the $IORB macro call. (Section 5 describes request blocks in general, Appendix D describes the IORB in detail, and Volume II describes the $IORB macro call.)

OVERLAY HANDLING FUNCTIONS

Overlay handling calls locate, load, execute, and unload fixed and floatable overlays. Fixed overlays are loaded into memory at a displacement from the base of the root segment fixed at the time of linking. Floating overlays are loaded as follows: If a bound unit can be shared between task groups (i.e., is linked as globally sharable), its floating overlays are loaded into system memory; otherwise, floating overlays are loaded into any sufficient block of memory in the memory pool of the issuing task's task group.

When bound units with fixed overlays are loaded, enough space is reserved in memory so that the linked, fixed overlay with the highest address can be loaded. Overlay handling calls similarly reserve overlay areas for floating overlays. Overlay areas are areas in memory of fixed size that accommodate the largest floating overlay associated with a bound unit. Overlay areas are managed by means of overlay area tables (OATs), which ensure that space in overlay areas is occupied only by overlays that are currently in use. Thus, overlay handling functions relieve the user of writing an overlay manager.

The overlay handling macro calls are:

- Overlay, Release, Wait, and Recall $OVRCL
- Overlay Area, Release $OVRLS
- Overlay Area, Reserve, and Execute Overlay $OVRSV
- Create Overlay Table $CROAT
- Delete Overlay Table $DLOAT
- Overlay, Execute $OVEXC
- Overlay, Load $OVLD
- Overlay, Status $OVST
- Overlay, Unload $OVUN

PHYSICAL I/O FUNCTIONS

The Request I/O ($RQIO) macro call, used in conjunction with the input/output request block (IORB), allows direct control by the user of device drivers or communication line protocol handlers. If direct access to devices is not a requirement, File System macro calls provide a more convenient means of handling input/output operations.

See Sections 6 and 7 for a complete description of physical I/O functions, including details on device drivers and line protocol handlers.

The macro routine/call for physical I/O is:

Request I/O Transfer	$RQIO

REQUEST AND RETURN FUNCTIONS

The macro calls for request and return functions enable an issuing task to perform the following:

- Ascertain the address of the first request block in the queue of requests placed against it
- Ascertain the completion status of request blocks placed against it
- Defer the processing of a request placed against it
- Terminate the request that it is processing, marking it as completed
- Wait for the completion of its own request(s) before resuming execution
- Issue a common return sequence for called subroutines.

When a task defers the processing of a request placed against it, it dequeues the request and requeues it at a specified priority level on its request queue. (This priority level is not to be confused with the priority level, or interrupt level, at which the task is running.) The deferred request is requeued at either the head or tail of any other requests deferred at the specified priority level. The capability of deferring a request is typically used by device drivers in order to give precedence to one type of request over another type.

The macro calls for request and return functions are:

• Dequeue and Post	$DQPST
• Defer Request on Tail	$DFRTL
• Defer Request on Head	$DFRHD
• Postpone Request on Tail	$PPNTL
• Recall from Head	$RCLHD
• Return Request Block Address	$RBADD
• Return	$RETRN
• Terminate Request	$TRMRQ
• Test Completion Status	$TEST
• Wait Any	$WAITA
• Wait for Operation to Complete	$WAIT
• Wait on Request List	$WAITL
• Wait on Multiple Request List	$WAITM

Section 5 and Volume II describe the macro calls for generating request blocks. Appendix D shows request block formats.

TERMINAL CONTROL FUNCTIONS

Terminal control functions allow secondary logins and the transfer of primary or secondary users between task groups.

When someone logs into the system as a secondary user, the Listener component attaches a secondary user's terminal to an existing task group if the user, when logging in, specifies the task group and if that task group has requested a secondary terminal.

The macro calls for terminal control functions permit:

- The task group to request any secondary terminal
- The task group to request a specific secondary terminal
- The task group to transfer a user to Listener, along with a new login line that automatically associates the user with another task group
- The task group to transfer a user, along with a new login line, to Listener, which later returns the user to the task group
- The task group to release a secondary terminal.

The appropriate macro calls are:

- Request Specific Terminal $RQSPT
- Request Terminal $RQTML
- Release Terminal $RLTML
- Transfer and Return User $XRETU
- Transfer User $XFERU

SEMAPHORE FUNCTIONS

A semaphore is a mechanism for coordinating the use of resources within task groups. Once defined, semaphores control access to multiple resources and control multiple requests for the same resource.

A semaphore is defined for each resource to be controlled and is given a 2-character ASCII semaphore name, which is a system symbol recognized by the Monitor. Every requestor of a resource whose use must be coordinated issues appropriate Monitor calls to the named semaphore to request or release the resource. The task that defines the semaphore assigns the semaphore's initial value. The monitor increments or decrements this initial value when the resource is released or requested/reserved, respectively. A requestor obtains use of a resource if the semaphore value is greater than zero at the time of the request. If the value is zero or negative, the requestor either waits until the resource becomes available or continues executing, depending upon the

macro call issued to make the request. The initial value of the semaphore determines the number of users who can utilize a resource at a given time. An initial value of 2 allows two simultaneous users, an initial value of 4 allows four users, etc.

Semaphore function macro calls are used to:

- Define a semaphore and set its initial value
- Increment the current-value counter
- Decrement the current-value counter
- Queue a semaphore request block if the requested resource is not available
- Remove a semaphore request block from its queue
- Delete a semaphore.

The macro calls for semaphore handling are:

- Cancel Semaphore Request $CNSRQ
- Define Semaphore $DFSM
- Release Semaphore $RLSM
- Request Semaphore $RQSM
- Reserve Semaphore $RSVSM
- Delete Semaphore $DLSM

STANDARD SYSTEM FILE I/O FUNCTIONS

A task group can access standard system files (command-in, user-in, user-out, error-out, and message library) through standard system file I/O macro calls. Other macro calls shown below allow the task to redefine certain standard system files. Specifically, the macro routines enable a task to:

- Read the next record from the command-in file
- Write the next record to the error-out file
- Read the next record from the user-in file
- Write the next record to the user-out file
- Redefine the user-in file
- Redefine the user-out file
- Redefine the message library file.

The macro calls are:

- Command In (read command-in file) $CIN
- Error Output File $EROUT
- New Command In $NCIN
- New Message Library File $NMLF
- New User Input File $NUIN
- New User Output File $NUOUT
- User Input File $USIN
- User Output File $USOUT

TASK CONTROL FUNCTIONS

The macro calls for task control allow the user to:

- Cancel a previously issued request
- Create, request, spawn, suspend, activate, delete, and abort a task
- Attach, load, transfer, and detach a bound unit to/from a task
- Create and delete a segment for a task's bound unit
- Process command lines
- Roll back (recover) updated records in all files updated since the last execution of Clean Point
- Declare a "clean point" at which
  - Updates made to records are complete
  - Updated records are written to disk
  - The updated file is considered to be in a consistent state
  - Records previously locked by the issuing task are unlocked.

Macro calls for task control are:

• Cancel Request	$CANRQ
• Clean Point	$CLPNT
• Command Line, Process	$CMDLN
• Create Segment	$CRSEG
• Delete Segment	$DLSEG
• Create Task	$CRTSK
• Delete Task	$DLTSK
• Request Task	$RQTSK
• Spawn Task	$SPTSK
• Bound Unit, Attach	$BUAT
• Bound Unit, Load	$BULD
• Bound Unit, Detach	$BUDT
• Bound Unit, Transfer	$BUXFR
• Kill Task	$KILLT
• Roll Back	$ROLBK

TASK GROUP CONTROL FUNCTIONS

A task group is a named set of one or more tasks, memory space, files, peripheral devices, and priority levels. Any number of task groups may be defined. (Task groups and tasks are explained in detail in the System Concepts manual.)

The macro calls for task control allow the user to:

- Create, spawn, request, or delete a task group
- Enable or disble certain functionalities (e.g., message chaining, ready prompt) for a task group
- Terminate a current task group and restart a task group request
- Abort a task group request
- Terminate a user session
- Declare a checkpoint from which processing can be restarted after premature termination of a group request
- Assign or disassign checkpoint files to a task group
- Abort a task group
- Terminate a user session.

A task executing under one group can initiate another group. First, a task group must be defined in order to create task group control structures and load the bound-unit root segment as the lead task. Then, a group request must be issued to activate the lead task for execution. Tasks can be executed concurrently in this task group with the use of control functions or commands.

The task group can be deleted; no more requests can be made against this group after it has been marked for deletion. When all tasks in the group terminate and become dormant, all memory associated with the group is returned to its memory pool, becoming available to other groups.

The several phases of task creation, activation, and deletion occur in sequence when a Spawn Task Group macro call is issued.

A task can suspend a task group's execution and then activate that task group.

A task can terminate the current group request and then restart the processing of the original task group request.

A task can abort the current request for the activation of a specified group. In this case, the next request (if any) against that group will be processed.

Aborting a task group deletes the group immediately, before all its tasks terminate and become dormant.

A task can terminate a user session, then either restart the group request, begin a new login sequence, or disconnect the user terminal.

Some macro calls listed below use a parameter block, which extends the argument list of the task request block. The macro call that generates parameter blocks ($PRBLK) is described in Volume II; block format is shown in Appendix D.

The macro calls for task group control are:

- Abort Group $ABGRP
- Abort Group Request $ABGRQ
- Activate Group $ACTVG
- Checkpoint $CKPT
- Checkpoint File $CKPFL
- Create Group $CRGRP
- Delete Group $DLGRP
- New Process $NPROC
- Request Group $RQGRP
- Set Group Attributes $SGRPA
- Spawn Group $SPGRP
- Suspend Group $SUSPG

TRAP HANDLING FUNCTIONS

The macro calls for trap functions allow an application to designate the traps to be handled during its execution. Specifically, the macro calls allow the user to:

- Connect a user-written, generalized trap handling routine to a task
- Enable a specific trap or all traps
- Disable a specific trap or all traps.

Additionally, the user can transmit a software-generated trap condition to a specific task.

Appendix A describes traps and trap handling in detail.

The macro calls for trap handling are:

- Disable User Trap $DSTRP
- Enable User Trap $ENTRP
- Trap Handler Connect $TRPHD
- Signal Trap $SGTRP

USER REGISTRATION FUNCTIONS

User registration functions enable a user to be registered in one or more subsystems, such as forms processing or networking. These functions create, retrieve, modify, and delete a subsystem record that establishes the user's access to a subsystem and contains various statistics.

Before a user's subsystem record can be created, the user must be registered in the system (as distinct from the subsystem) by the system administrator. To register a user in the system, the administrator creates a registration record by means of the Edit Profile utility. One user registration function, Profile Record, Get User Information ($PRFIF), retrieves limited information from the registration record. The subsystem and registration records belong to the profiles file, which is the system's user registration data base.

Using the Edit and List Profile utilities, the system administrator can maintain a user's subsystem record(s) as well as registration record. First, however, the system programmer must build a subsystem module as an interface between the utilities and subsystem records. Specifications for subsystem modules are given in Appendix H.

User registration macro calls allow the user to:

- Create a skeletal subsystem record that contains user id, time of creation, and subsystem id
- Read a subsystem record
- Read limited information from a registration record
- Update a subsystem record
- Request and verify a password from the user of a terminal that has experienced a phsyical disconnection.

User registration macro calls are:

• Profile Record, Accounting Update	$PRFAU
• Profile Record, Create	$PRFCR
• Profile Record, Delete	$PRFDL
• Profile Record, Get	$PRFGT
• Profile Record, Get User Information	$PRFIF
• Profile Record, Update	$PRFUP
• Reverify Password	$RVFPW

SOFTWARE REBOOT

The Software Reboot Facility reinitializes the system without operator intervention. It is activated dynamically by exhaustion of trap save areas or indirect request blocks, and by Watchdog Timer timeouts. The user can direct that a dump be taken before reinitialization of the system.

The Software Reboot routines/calls are:

• Modify Reboot Parameters	$RBPRM
• Reboot	$RBOOT
• Unlock Dumpfile	$RLDMP

Section 3
FILE SYSTEM FUNCTIONS

File system macro calls enable applications to access data files, including device files. These functions fall into the following categories:

- File management
- Data management
- Storage management.

This section describes each category and its use of the File Information Block (FIB). All of the functions mentioned below are described in detail in Volume II of this manual.

FILE MANAGEMENT FUNCTIONS

The macro calls for file management consist of the following functions:

Associate File	$ASFIL
Change Working Directory	$CWDIR
Close File	$CLFIL
Create Directory	$CRDIR
Create File	$CRFIL
Delete File	$DLFIL
Delete Directory	$DLDIR
Dissociate File	$DSFIL
Expand Pathname	$XPATH
Get Device Information	$GIDEV

Get File	$GTFIL
Get File Access Rights	$GAFIL
Get File Accounting Information	$GTACT
Get File Information	$GIFIL
Get Working Directory	$GWDIR
Grow File	$GRFIL
Open File	$OPFIL
Remove File	$RMFIL
Rename File/Directory	$RNFIL
Modify File	$MDFIL
Set Terminal File Characteristics	$STTY
Test File For Input	$TIFIL
Test File For Output	$TOFIL
Shrink File	$SHFIL
Swap File	$SWFIL
Wait For File Input	$WIFIL
Wait For File Output	$WOFIL
Cleanpoint	$CLPNT
Rollback	$ROLBK.

The macro calls listed above are preparatory to processing a file. Specifically, file management macro calls allow the user to perform the following:

- Create a file
- Delete a file
- Get a file (reserve a file for processing)
- Open a file
- Close a file
- Remove a file from processing
- Rename a file
- Modify a file's attributes
- Associate a logical file number with a pathname
- Dissociate a logical file number from a pathname
- Create a directory
- Delete a directory
- Rename a directory
- Change the working directory
- Get the name of the current working directory

- Expand disk space allocated to a file
- Contract disk space allocated to a file
- Expand pathname (develop a full pathname from a relative pathname)
- Get information about a file
- Test the status of an I/O activity (terminal)
- Wait for the completion of an asynchronous I/O activity (terminal)
- Set the file characteristics of a terminal
- Return (recover) a file to its last consistent state after a system or software failure
- Swap to the next section of a multivolume tape file or disk serial multivolume file.

Although the following functions are available through macro calls, they are typically performed outside of program execution by means of execution control (ECL) commands:

- Get File
- Remove File
- Create File
- Delete File
- Grow File
- Shrink File
- Rename File
- Modify File
- Create Directory
- Delete Directory
- Change Working Directory
- Get Working Directory
- Set Terminal File Characteristics
- Associate File
- Dissociate File.

DATA MANAGEMENT FUNCTIONS

The following macro calls are considered data management functions:

Delete Record	$DLREC
Read Record	$RDREC
Rewrite Record	$RWREC
Write Record	$WRREC.

The above macro calls provide for the transfer of logical records between the user's record storage area and external files. Before any data management calls can be executed, the file to be accessed must have been reserved (by means of the Get File or Create File functions) and opened (by means of the Open File function). Moreover, before a file can be opened, it must have been associated with a logical file number (LFN) by means of an Associate File, Get File, or Create File function. Thus, data management and file management macro calls are interdependent. Figure 3-1 partially illustrates this interdependence.

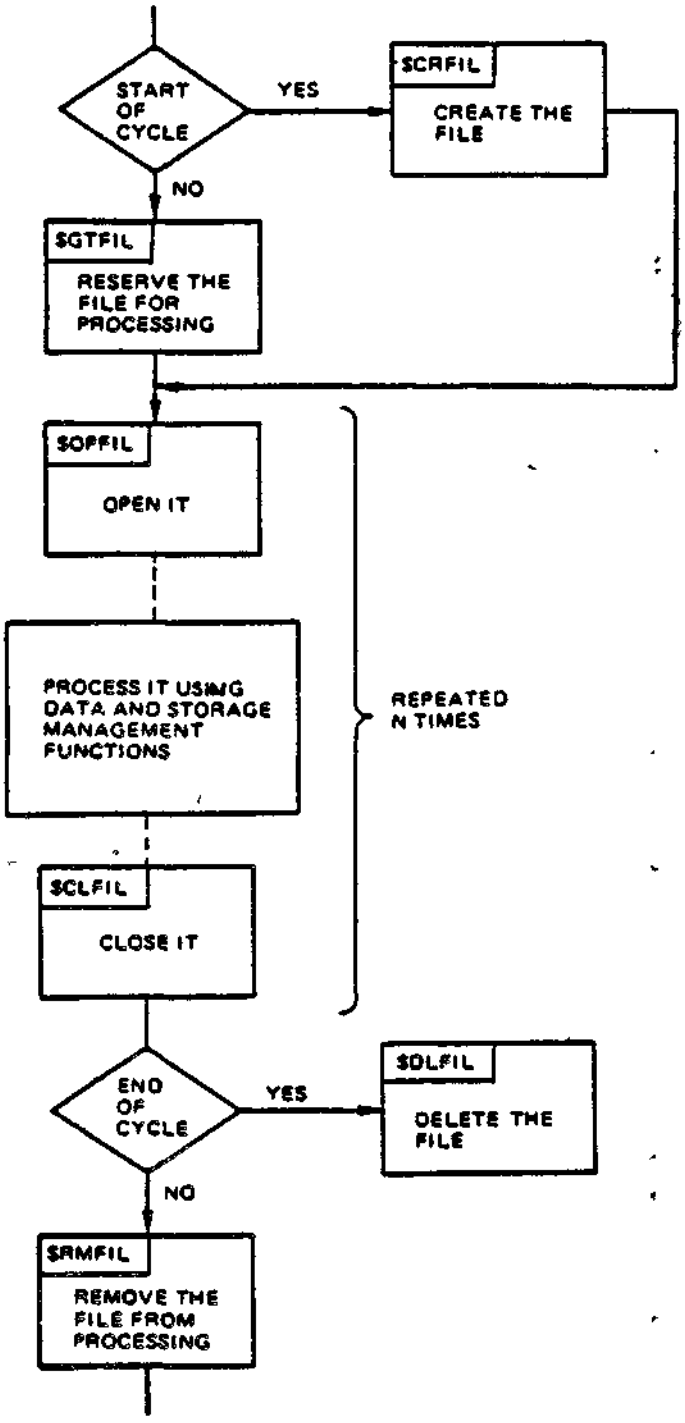


Figure 3-1. Life Cycle of a File

STORAGE MANAGEMENT FUNCTIONS

The following macro calls perform storage management functions:

Read block	$RDBLK
Wait block	$WTBLK
Write block	$WRBLK.

These calls transfer physical blocks of data between the user's buffer and an external file. Storage management itself is used transparently by data management to perform input/output. An initial Read Record ($RDREC) call, for example, causes storage management to transfer a block of data from external storage to a buffer in memory. Data management then unblocks a record and transfers it to a second buffer within the application.

By means of storage management read and write functions, the user can transfer blocks of data directly to or from an application buffer, bypassing an intermediate buffer and the blocking/deblocking operations performed by data management. Although highly efficient, storage management places on the user responsibility for observing various file organizations and formats while blocking/deblocking. The user of storage management must also provide any necessary control information, such as control interval headers and logical record headers.

By creating two application buffers and by using the Wait Block macro call (described in Volume II) the user can perform asynchronous I/O (i.e., process one block of data while another is being transferred from device to memory).

Like data management macro calls, storage management macro calls cannot be executed until the file to be accessed has been reserved, opened, and associated with an LFN.

FILE INFORMATION BLOCK

Data management, storage management, and several file management functions must pass arguments to the file system by means of a data structure called the File Information Block (FIB). The arguments passed include the LFN of the file to be accessed, the address of the user's record area, the size of input and output records, and the type of key by which records are to be located.

The following macro calls must use an FIB:

Open File	$OPFIL
Close File	$CLFIL
Swap File	$SWFIL
Test File	$TIFIL, $TOFIL
Read Record	$RDREC
Write Record	$WRREC

Rewrite Record	$RWREC
Delete Record	$DLREC
Read Block	$RDBLK
Write Block	$WRBLK
Wait Block	$WTBLK

Some of the arguments required for one type of macro call (e.g., storage management) are not applicable to the other types. Thus, a FIB generated for data/file management functions differs in format from a FIB generated for storage management functions.

The user can generate a FIB and and values for its entries by means of the $FIB macro call. Depending on the argument(s) supplied with it, $FIB does one of the following:

- Generates an FIB, with default values, for data/file management
- Generates an FIB for data/file or storage management, with values defined by the user
- Modifies values of an existing FIB.

Using $FIB, the user can set values for a new or existing FIB by means of keywords that specify a field and expressions that specify a value. The $FIB argument "IRL=90", for example, refers to the input record length field of a data/file managment FIB and sets a maximum input record length of 90 bytes. Other keywords are specific to storage management functions.

To modify the fields of an existing FIB, the user can employ offset tags rather than $FIB keywords. (Offset tags are discussed later in this section and in Section 5). $FIBDM generates tags specific to data/file management functions; $FIBSM generates tags specific to storage/management functions. $TFIB generates two sets of tags applicable to both kinds of file system functions.

File Information Block (FIB) for Data Management

Table 3-1 describes the entries of a FIB used with data/file management macro calls. The offset tags for these entries, generated by $FIBDM, are shown in Appendix D.

Table 3-1. File Information Block (FIB) for Data Management

Entry	Size (bytes)	Description
Logical file number (LFN)	2	Specifies the logical file number (LFN) by which the file is referenced. The LFN is the common element linking the FIB and the external file; this connection is made via the $ASFIL, $CRFIL, or $GTFIL macro call (or equivalent command).
Program view	2	Describes user visibility to the file, and the file's functional capabilities. Bit 0 set to 0 indicates that this FIB is to be used for data management (record level) access. Table 3-2 describes this entry in detail and its bit settings for data management macro calls.
User record pointer	4	Identifies the start of the user-record area as follows: $RDREC - Identifies the storage area into which records are delivered by the system. $RWREC, $WRREC - Identifies the storage area from which records are taken by the system. The storage area must be large enough to contain the longest record, excluding headers, to be written to or received from the file.
In record length	2	Specifies the maximum size (in bytes) of the user-record area for $RDREC operations.

Table 3-1 (cont). File Information Block (FIB) for Data Management

Entry	Size (bytes)	Description
Out record length	2	Specifies the actual size (in bytes) of the record to be written or read, as follows: $RDREC - The system updates this entry to reflect the actual length (in bytes) of the last record delivered into the user-record area. $RWREC, $WRREC - Specifies the actual length (in bytes) of the record, excluding the headers, to be written in the file.
In record status	1	On write operations, indicates the type of terminal control information in each record as follows: 0000 = unknown terminal control information 0001 = no terminal control information 0010 = standard GCOS 6 printer control characters
Out record status	1	On read-record operations bit 0 = 1 indicates that the record just read is a duplicate of a previous record (i.e., it contains the same key value as the previous record). On write-record or rewrite-record operations bit 0 = 1 indicates that the record just written is a duplicate (i.e., it contains the same key value as a record already in the file). On read-record operations bit 1 = 1 indicates that there are more duplicates for this record still remaining in the file.

Table 3-1 (cont). File Information Block (FIB) for Data Management

Entry	Size (bytes)	Description
Out record status (cont)		For example, if three records exist with the same key value, then reading the first one will return in this entry: bit 0 = 0 bit 1 = 1; reading the second record will return: bit 0 = 1 bit 1 = 1; reading the last record will return: bit 0 = 1 bit 1 = 0
In record type	2	$RDREC - Specifies the record type of the record to be read. 'FFFF' indicates that any record type is acceptable.
Out record type	2	$WRREC, $RWREC, $DLREC - Specifies the record type of the record to be updated. $RDREC - Specifies the record type of the record delivered to the user.
In key pointer	4	Identifies the start of the user-key area in which the key value is stored for the following $RDREC macro call functions: Read with key Read position equal Read position greater than Read position greater than or equal Read position forward Read position backward

Table 3-1 (cont). File Information Block (FIB) for Data Management

Entry	Size (bytes)	Description
In key pointer (cont)		For the following $WRREC macro call functions: Write with key Write position equal Write position greater than Write position greater than or equal Write position forward Write position backward For the following $RWREC macro call function: Rewrite with key And for the following $DLREC macro call function: Delete with key For CALC, Primary, and Alternate keys, the keys to be used must be initialized within the user's record area and the field must point to that key. The type of key is specified in the "in key format" entry below.

Table 3-1 (cont). File Information Block (FIB) for Data Management

Entry	Size (bytes)	Description
In key format	1	Identifies the type of key pointed to by the "in key pointer" entry above, as follows: 0 - None specified; the type of key is determined by the format of the file. 1 - Primary, relative, or CALC (Random), as determined by the file format: o Primary key for indexed files o Relative key for relative files o CALC key for random files 2 - Simple key 3 - Alternate key -1 - Current key of reference The entry is meaningful only for the macro calls specified in the "in key pointer" entry defined above.
In key length	1	Specifies the length (in bytes) of the user-key area identified in the "in key pointer" entry described above. Only meaningful for primary, alternate, and CALC keys; simple and relative keys are always assumed to be four bytes.
Out record address	4	This field is available for the system to place the media address of the last record transferred by the last data management macro call.

Table 3-1 (cont). File Information Block (FIB) for Data Management

Entry	Size (bytes)	Description
Out record address (cont)		Normally, this address is a 32-bit simple key (i.e., it specifies the control interval and logical record number within the control interval). However, if the file is accessed via a relative key as specified in the "in key format" field, then this address is a 32-bit relative key (i.e., relative logical record number in the file). This field is undefined if the operation is not performed as expected. For card readers, printers, and terminal devices, this field contains a count of the records transferred; i.e., this field is incremented by 1 for each access to the device.
Reserved	4	This entry is reserved for future use; must be set to zeros.

Program View Entry in FIB for Data Management

Table 3-2 shows the contents of the 2-byte program view entry for data management (record level) access. The program view entry describes to the file system how the file is to be accessed, and, to some extent, what it looks like to the programmer. The file system uses the FIB's contents to ensure that the file is accessed only as intended. Keywords of the $FIB macro call and offset tags generated by $FIBDM both provide a means of refering to fields within the program view entry.

Bits 0 through 9 of the program view entry are processed only when the file is opened, and cannot be changed while the file is open.

Table 3-2. Program View Entry in FIB for Data Management

Entry	Size (bits)	Description	Related Macro Calls
Access level (Bit 0)	1	Specifies that file is accessed via data management macro calls, as follows: 0 - Access via data management macro calls.	$OPFIL
Process rules (Bits 1-4)	4	Specifies how the file can be processed; that is, it specifies which types of data management macro calls are allowed as follows: Binary — Permitted Macro Calls 1000 — $RDREC 0100 — $WRREC 0010 — $RWREC 0001 — $DLREC nnnn — Any combination of the settings to allow the desired data management macro calls listed above. A macro call that is not permitted (as specified in this field) causes an access violation error.	

Table 3-2 (cont). Program View Entry in FIB for Data Management

Entry	Size (bits)	Description	Related Macro Calls
Key type (Bits 5-9)	5	Specifies the type of keys that can be used to access the file as follows: Binary — Permitted Key Type 10000 Primary 01000 CALC (Random) 00100 Alternate 00010 Relative 00001 Simple 00101 Alternate and Simple 10101 Alternate and Simple plus Primary 01101 Alternate and Simple plus CALC 00111 Alternate and Simple plus Relative If the key type specified in this field is not permitted by the type of file being processed, a bad program view error results. The following types of keys are allowed by the specified types of files:	$OPFIL

Table 3-2 (cont). Program View Entry in FIB for Data Management

Entry	Size (Bits)	Description	Related Macro Calls
Key type (bits 5-9) (cont)		File Organization — Key Type UFAS Indexed, Alternate — Primary UFAS Random — CALC UFAS Disk Resident Files — Alternate UFAS Relative, Fixed Relative — Relative UFAS Disk Resident Files — Simple	
Record class (bit 10)	1	Specifies type of logical records that can be present in the file as follows: 0 - Any type (i.e., fixed- or variable-length records allowed). 1 - Only fixed-length records allowed.	$RDREC $WRREC $RWREC
Record visibility (Bit 11)	1	Specifies whether or not deleted records are skipped during read next record ($RDREC) operations as follows: 0 - Deleted records not visible (i.e., skip them) 1 - Deleted records are visible (i.e., the system issues the record not found return code when a deleted record is accessed).	$RDREC

Table 3-2 (cont). Program View Entry in FIB for Data Management

Entry	Size (Bits)	Description	Related Macro Calls
Key storage area alignment (Bit 12)	1	Specifies the boundary alignment of the user-key area (see "in key pointer" entry in Table 3-1) as follows: 0 - Key storage area begins at even-byte boundary (word-aligned). 1 - Key storage area begins at odd-byte boundary.	$RDREC $WRREC $RWREC $DLREC
Record storage area alignment (Bit 13)	1	Specifies the boundary alignment of the user-record area (see "User Record Pointer" entry in Table 3-1) as follows: 0 - Record storage area begins at even-byte boundary (word-aligned). 1 - Record storage area begins at odd-byte boundary.	
Transcription mode (Bit 14)	1	Specifies how data is to be transferred as follows: 0 - Data is transferred in device-specific native (ASCII) mode. 1 - Data is transferred in binary transcription mode. (See Note 2.)	$RDREC $WRREC
Reserved (Bit 15)	1	Reserved; must be zero.	None

NOTES

1. Bits 10 through 15 may be set after an $OPFIL macro call and before any data management macro call.

Table 3-2 (cont). Program View Entry in FIB for Data Management

2. Binary transcription mode is meaningful only for card devices, seven-track tapes, and EBCDIC tapes. For card devices, this mode is equivalent to verbatim mode (see Section 6).

File Information Block (FIB) for Storage Management Access

Table 3-3 describes the entries of a FIB used with storage management macro calls. The offset tags for these entries, generated by $FIBSM, are shown in appendix D.

Table 3-3. File Information Block (FIB) for Storage Management

Entry	Size (bytes)	Description
Logical file number (LFN)	2	Specifies the logical file number with which the file is referenced. The LFN is the common element linking the FIB with the external file; this connection is made with the $ASFIL, $CRFIL, or $GTFIL macro call, or equivalent command.
Program view	2	Describes the user visibility to the file and the file's functional capabilities. Bit 0 set to 1 indicates that this FIB is to be used for storage management (block level) access. Table 3-4 describes this entry in detail, and its bit settings for storage management macro calls.
Buffer pointer	4	Identifies the start of the buffer area as follows: $RDBLK - Identifies the buffer area into which blocks of data are delivered. $WRBLK - Identifies the buffer area from which blocks of data are taken.
Transfer-size	2	Specifies the size (in bytes) of the data transfer (i.e., the size of the buffer).

Table 3-3. File Information Block (FIB) for Storage Management

Entry	Size (bytes)	Description
Block size	2	Specifies the size of the block (in bytes). For disk files the size must be a multiple of physical sector size.
Block number	4	Specifies the starting block number for the I/O transfers; is relative to the start of the file and to the block size (described above). This entry is incremented by 1 after each I/O transfer; therefore, user's dynamic changes to the block size also require changes to the contents of this entry. The first block in a file is block 0.
Reserved	16	Reserved for later use; must be set to zeros.

Program View Entry in FIB for Storage Management

Table 3-4 shows the contents of the 2-byte program view entry for storage management (block level) access. The program view entry describes to the file system how the file is to be accessed, and to some extent, what it looks like to the programmer. The file system uses the FIB's contents to ensure that the file is accessed only as intended. Keywords of the $FIB macro call and offset tags generated by $FIBSM both provide a means of referring to fields within a program view entry.

Bits 0 through 9 of the program view entry are processed only when the file is opened, and cannot be changed while the file is open.

Table 3-4. Program View Entry in FIB for Storage Management

Entry	Size (bits)	Description	Related Macro Calls
Access level (Bit 0)	1	Specifies that file is accessed via storage management macro calls, as follows: 1 - Access via storage management macro calls.	$OPFIL
Process rules (Bits 1-4)	4	Specifies how the file can be processed; that is, it specifies which types of storage management macro calls are allowed as follows: Binary — Permitted Macro Calls 1000 — $RDBLK 0100 — $WRBLK 1100 — $RDBLK, $WRBLK A macro call that is not permitted in this field causes an access violation error.	
Reserved (Bits 5-12)	8	Reserved; must be set to zeros.	
Buffer Alignment (Bit 13)	1	Specifies the boundary alignment of the user buffer (see "Buffer Pointer" in Table 3-3) as follows: 0 - Buffer begins at even-byte boundary (word aligned).	$RDBLK $WRBLK

Table 3-5 (cont). Offsets Definition Macro Calls

Macro Call	Affected Structure
$TFIB	File information block for the following macro calls: Open File $OPFIL Close File $CLFIL Test File $TIFIL, $TOFIL Read Record $RDREC Write Record $WRREC Rewrite Record $RWREC Delete Record $DLREC Read Block $RDBLK Write Block $WRBLK Wait Block $WTBLK
$FIBDM	File information block specific to data management (record level) access; used for the following macro calls: Open File $OPFIL Close File $CLFIL Test File $TIFIL, $TOFIL Read Record $RDREC Write Record $WRREC Rewrite Record $RWREC Delete Record $DLREC
$FIBSM	File information block specific to storage management (block level) access; used for the following macro calls: Open File $OPFIL Close File $CLFIL Read Block $RDBLK Write Block $WRBLK Wait Block $WTBLK
$MDPSB	Argument structure for Modify File macro call ($MDFIL)

Offsets definition macro calls can be specified only once per assembly procedure. They provide tags that are equated to specific offsets in argument structures and FIBs. For example, assuming that the address of an argument structure labeled FILE_A has been loaded into a base register as follows:

```
LAB  $B4,FILE_A
```

and assuming that $CRPSB has been specified, the following address syllable can be used to refer to the argument structure entry that identifies the control interval size:

```
$B4.R_CISZ
```

This entry effectively points to the displacement FILE_A+5 in the parameter structure.

Volume II of this manual describes each displacement definition macro routine/call and its tags, displacements, and entry names in detail.

Table 3-4 (cont). Program View Entry in FIB for Storage Management

Entry	Size (Bits)	Description	Related Macro Calls
Buffer Alignment (Bit 13) (cont)		1 - Buffer begins at odd-byte boundary.	
Transcription mode (Bit 14)	1	Specifies how data is transferred as follows: 0 - Data is transferred in device-specific native (ASCII) mode. 1 - Data is transferred in binary transcription mode. (See Note 2.)	$RDBLK $WRBLK
Synchronous/ asynchronous indicator (Bit 15)	1	Specifies whether or not $RDBLK or $WRBLK macro calls are executed synchronously or asynchronously as follows: 0 - $RDBLK or $WRBLK macro calls are to be executed synchronously. When synchronous $RDBLK or $WRBLK macro calls are issued, a $WTBLK macro call is not required to synchronize buffer use. 1 - $RDBLK or $WRBLK macro calls are to be executed asynchronously (i.e., a $WTBLK macro call is required to synchronize.)	$RDBLK $WRBLK

NOTES

1. Bits 10 through 15 may be set after an $OPFIL macro call and before any Storage Management macro call.

2. Binary transcription mode is meaningful only for card devices, seven-track tapes, and EBCDIC tapes. For card devices, this mode is equivalent to verbatim mode (see Section 6).

Offsets Definitions

You can refer to specific locations in the file information block and other argument structures by using offsets definition macro calls. These calls, summarized in Section 5 and described in detail in Volume II of this manual, define offsets tags.

Table 3-5 shows the offsets definition macro calls and the structures for which they define tags.

Table 3-5. Offsets Definition Macro Calls

Macro Call	Affected Structure
$CRPSB	Argument structure for Create File macro call ($CRFIL)
$CRKDB	Key descriptor block pointed to by the $CRPSB argument structure
$DIPSB	Argument structure for Get Device Information macro call
$GTPSB	Argument structure for Get File macro call ($GTFIL)
$GAPSB	Argument structure for Get File Access Rights macro call ($GAFIL)
$GIPSB	Argument structure for Get File Information macro call ($GIFIL)
$GRPSB	Argument structure for Grow File macro call ($GRFIL)
$GIFAB	File attribute block pointed to by the $GIPSB argument structure
$GIKDB	Key descriptor block pointed to by the $GIPSB argument structure
$SHPSB	Argument structure for Shrink File macro call ($SHFIL)

Section 4
COMMUNICATIONS PROCESSING FUNCTIONS

Communications processing refers, in this section, to the transfer of data between an application program and a remote device (i.e., terminal or printer). A remote device is one connected to a Multiline Communications Processor (MLCP); a local device is attached instead to a Multiple Device Controller (MDC). The control of local devices by means of device drivers is discussed in Section 6.

OVERVIEW OF COMMUNICATIONS PROCESSING

The user can control the transfer of data between an application program and a remote device either by means of the file system or, more directly, by physical input/output.

Using the file system, the programmer employs many of the file system functions described in Section 3 (e.g., Open File, Read Record, Write Record). The parameters for these operations are passed between the application program and the file system by means of the file information block (FIB), which is also described in Section 3. The system translates the values of FIB entries into values for the entries of the input/output request block (IORB). Thus marked, the IORB provides instructions to a line protocol handler (LPH), which carries out the desired input/output operation.

Using physical I/O, the programmer directly constructs and issues the IORB instead of doing so indirectly by means of file system functions and the FIB. To write output to a terminal, for example, the programmer performs the following:

1. Generates an IORB by means of the $IORB macro call

2. Generates IORB offsets tags (by means of the $IORBD macro call), which enable the programmer to refer to and fill fields in the IORB

3. Sets, in the appropriate IORB fields, a write function code and parameters specializing the write operation

4. Issues a Request Input/Output ($RQIO) macro call, which causes the appropriate LPH to perform the operation indicated by the IORB.

The above example assumes that the device being written to has already been connected by means of previously issued $IORB and $RQIO macro calls (as explained in the final subsection).

COMMUNICATIONS PROCESSING THROUGH THE FILE SYSTEM

The following subjects are discussed below:

- File system functions applicable to the communications processing
- Synchronous and asynchronous I/O
- Use of specific file system functions
- Sequences of file system functions useful for communications processing
- Use of the Set Terminal Characteristics (STTY) function/command for changing terminal characteristics.

File System Functions

The file system functions applicable to communications processing fall under the headings of File Management and Data Management.

FILE MANAGEMENT FUNCTIONS

By means of these functions, a terminal can be reserved for processing, opened, closed, and associated or dissociated with a logical file number (LFN) that identifies the file to the system. The macro calls that perform these and other related functions are:

Get File	$GTFIL
Open File	$OPFIL
Close File	$CLFIL
Associate File	$ASFIL
Dissociate File	$DSFIL
Test File	$TIFIL/$TOFIL
Wait File	$WIFIL/$WOFIL

DATA MANAGEMENT FUNCTIONS

Data management functions enable an application to read and write logical records either synchronously or asynchronously. (Synchronous and asynchronous I/O operations are explained later in this section.) Data management functions are:

Read Record	$RDREC
Write Record	$WRREC

Synchronous Input/Output

A terminal can be configured for either synchronous or asynchronous I/O operations. In synchronous operations, the processing of data and the transfer of data (between application and terminal) occur sequentially rather than simultaneously. Thus, the application must wait until the transfer of data is complete before processing can resume. Synchronous I/O is best suited to situations in which data is transferred between an application and a single terminal and to such activity as connecting and disconnecting a terminal.

Asynchronous Input/Output

If a terminal is configured for asynchronous I/O, data is transferred between the terminal and the application by way of a system buffer. Thus, asynchronous I/O allows the application to process records while the file system reads or writes records to or from the buffer.

Asynchronous I/O and the data/file management functions listed above allow an application to access multiple interactive terminals efficiently. For terminals operating asynchronously, the system automatically schedules an anticpatory read, which transfers input entered at the terminal to a buffer in system memory. If an application immediately issues a Read Record ($RDREC) call, the task must wait until the system buffer has received input from the terminal. While the task is waiting, data may be available from another terminal reserved by the application. Instead, the application can issue the Test Input File ($TIFIL) macro call to determine whether a read has completed at a specific terminal. Alternatively, Wait File for Input ($WIFIL) can be used to wait until a read has completed at any of the reserved terminals. A subsequent Read Record to the terminal would then return the data for processing by the application. The Test File function also enables an application to test the completion of a physical connection to a terminal before issuing an order to that terminal.

Using File System Functions

This subsection provides specific information on the use of the following data and file management functions:

Get File	$GTFIL
Open File	$OPFIL
Test File	$TIFIL/$TOFIL
Wait File	$WIFIL/$WOFIL

GET FILE ($GTFIL) MACRO CALL GUIDELINES

The Get File function reserves a file for processing and connects a file to a logical file number (LFN). The LFN is used in other file system calls (e.g., $OPFIL, $RDREC, $WRREC) to refer to the file in question. Normally, the Get File function is invoked by a Get File command outside program execution.

The arguments for the Get File ($GTFIL) macro call in an Assembly language communications program must have the values shown in Table 4-1.

Table 4-1. Arguments for Get File ($GTFIL) Macro Call

Argument	Argument Value
Logical file number (LFN)	A value from 0 through 255
Pathname pointer	Must point to a pathname of a communications device (e.g., !TTY01)
Concurrency control	According to how the application uses the device (normally zero for exclusive use)
Remaining arguments	Zero

OPEN FILE ($OPFIL) MACRO CALL GUIDELINES

The Open File function allocates buffer space (if required) and physically connects the device or terminal.

The Open File macro call $OPFIL, when used in communications, must include the location of the file information block (FIB), which in turn must contain a valid program view item.

TEST FILE ($TIFIL, $TOFIL) MACRO CALL GUIDELINES

Before the application issues a $RDREC macro call, it can issue the Test Input File ($TIFIL) macro call to check whether input is available.

Before the application issues a $WRREC macro call, it can issue the Test Output File ($TOFIL) macro call to check whether the preceding output operation was completed.

WAIT FILE ($WIFIL, $WOFIL) MACRO CALL GUIDELINES

The use of the Wait File macro call permits an application to wait for the completion of an outstanding read or write order. The Wait File macro call can be used with a set of terminals or devices. Test and Wait File macro calls differ in terms of when control is returned to the calling routine. A Test File call will return immediately with a busy or not busy status. An application would block the execution of lower level tasks with repeated test file calls to a busy file. This problem can be avoided by issuing a Wait File macro call in lieu of successive Test File macro calls.

$WIFIL is used to wait for input from any device/terminal; $WOFIL to wait for completion of output to any device/terminal.

Macro Call Sequences

This subsection describes sequences of file system macro calls commonly used by applications that access communications devices. Each sequence of macro calls applies to a different type of communications processing.

The types of communications processing illustrated below are:

- Input only (TTY or STD data entry applications)
- Output only (receive-only printer (ROP) application)
- Bidirectional (the device is opened either for input or output, but not both (BSC 2780))
- Interactive (TTY, STD, or BSC 3780 applications).

MACRO CALL PROCEDURES FOR DATA ENTRY TERMINALS

Table 4-2 shows the procedure for using file system macro calls in a communications application involving data entry terminals.

Table 4-2. Macro Call Procedures for Data Entry Terminals

Procedure Step	Action by Application Program	System Actions
1	Issue $GTFIL macro call.	
2	Issue $OPFIL macro call with FIB program view bit 1 set to 1, bit 2 set to 0.	Issues asynchronous connect; returns a normal status to the program.
3	Issue $WIFIL macro call to wait until connect is complete and input is available. (With multiple devices, the $WIFIL macro call can be issued with a list of LFNs, effectively giving up control until input is available from one or more devices in the list.)	Returns when a read has been satisfied.

Table 4-2 (cont). Macro Call Procedures for Data Entry Terminals

Procedure Step	Action by Application Program	System Actions
3 (cont)	Otherwise, if application is to do other processing (not giving up control), issue $TIFIL macro call.	If connect is not complete, returns a busy status. If connect is complete, issues an asynchronous read and returns a busy status until read is complete.
4	If not-busy status is returned, issue $RDREC macro call.	With read operation complete, moves data from system buffer to application's buffer, issues another asynchronous read, and returns a normal status to the program.
5	If an error status is returned, exit from the procedure.	
6	When read is successful, return to step 3 to request more data from the device.	
7	When application processing is completed, issues $CLFIL macro call.	Issues a disconnect.
8	Issue a $RMFIL macro call.	

MACRO CALL PROCEDURES FOR OUTPUT-ONLY TERMINALS

Table 4-3 shows the procedure for using macro calls in communications applications involving output-only terminals.

Table 4-3. Macro Call Procedures for Output-Only Terminals

Procedure Step	Action by Application Program	System Actions
1	Issue $GTFIL macro call.	
2	Issue $OPFIL macro call with FIB program view bit 1 set to 0, bit 2 set to 1.	Issues an asynchronous connect, returns a normal status to the program.
3	Issue $WOFIL macro call to wait until connect is complete and output can be transmitted. (With multiple devices, the $WOFIL macro call can be issued with a list of LFNs, effectively giving up control until output can be sent to one or more of the devices in the list.)	Will return when output can be transmitted.
	Otherwise, if the application is to do other processing (not give up control), issue a $TOFIL macro call.	If connect is not complete, returns a busy status. If connect is complete, returns a not busy status if output can be transmitted.
4	If not-busy status is returned, issue $WRREC macro call.	Moves data from application buffer to system buffer. Issues asynchronous write and returns a normal status to the application.
5	If error status is returned, exit from the procedure.	
6	When write is successful, return to step 3 to transmit more data to the device.	
7	When application processing is complete, issue $CLFIL macro call.	Issues disconnect according to device type.
8	Issue $RMFIL macro call.	

Macro Calls for a Single Interactive Terminal

Table 4-4 describes the procedures for using macro calls in communications applications involving only one interactive terminal that has been configured for non-buffered synchronous input/output operation.

Table 4-4. Macro Call Procedures for Single Interactive Terminal

Procedure Step	Action by Application Program	System Actions
1	Issue $GTFIL macro call.	
2	Issue $OPFIL macro call with FIB program view bit 1 set to 1, program view bit 2 set to 1.	
To read from the terminal and then write to the terminal:		
3	Issue $RDREC macro call. (This effectively gives up control until the read is satisfied.) If error status returned, exit from the procedure.	Data is read directly into the application buffer.
4	Process the data just read.	
5	Issue $WRREC. (This effectively gives up control until the write is complete.) If an error status is returned, exit from the procedure.	Data is written directly from the application buffer.
6	If additional input is expected, refer to step 3.	
7	When application processing is complete, issue $CLFIL macro call.	Issues a disconnect.
8	Issue $RMFIL macro call.	

MACRO CALL PROCEDURES FOR MULTIPLE INTERACTIVE TERMINALS

Table 4-5 describes the procedures for using macro calls in communications applications involving multiple terminals configured for buffered, asynchronous operation.

Figure 4-1 illustrates the procedure's flow.

Table 4-5. Macro Call Procedures for Multiple Terminals

Procedure Step	Action by Application Program	System Actions
1	Issue $GTFIL macro call to each terminal.	
2	Issue $OPFIL macro call to each terminal with FIB program view bit 1 set to 1, bit 2 set to 1.	Issues asynchronous connect; returns normal status to the program.
To read from a terminal and then write to a terminal:		
3	Issue $WIFIL macro call with a list of LFNs. (This will effectively give up control until input is available from one or more terminals in the list.)	Returns when a read is complete and data is available. Returns the LFN of the first terminal in the list for which data is available.
4	Issue $RDREC macro call.	Moves data from system buffer to application's buffer, issues another asynchronous read, and returns a normal status to the program.
5	If an error status is returned, exit from the procedure.	
6	Process the data just read.	

Table 4-5 (cont). Macro Call Procedures for Multiple Terminals

Procedure Step	Action by Application Program	System Actions
7	Issue $WRREC macro call. (This will give up control until output can be sent to terminal.)	Waits until output can be sent, moves data from the application's buffer to system buffer, and issues an asynchronous write.
8	If additional input is expected from any terminal, see step 3.	
9	When application processing is complete, issue $CLFIL call.	Issues disconnect.
10	Issue $RMFIL macro call.	

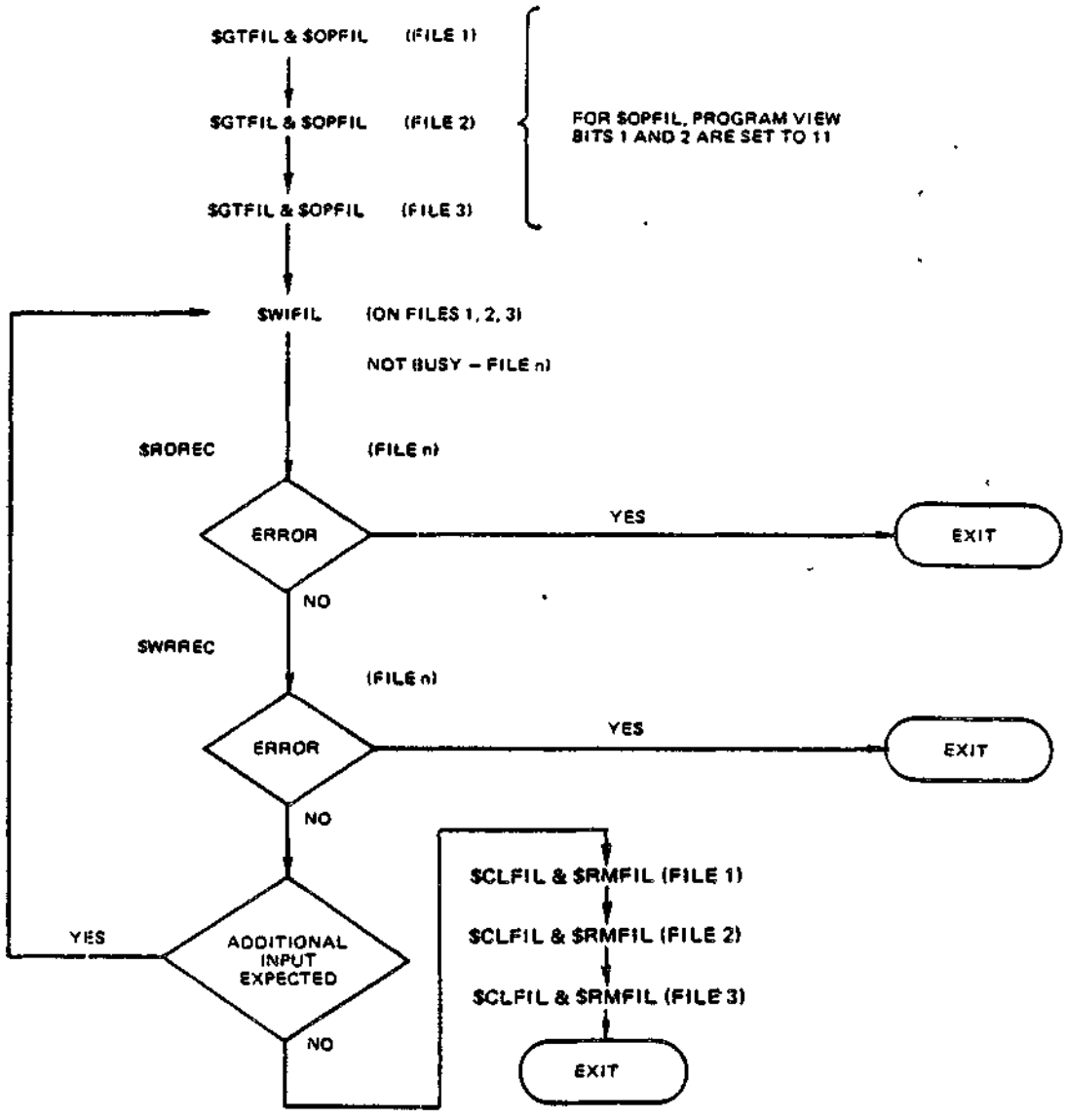


Figure 4-1. Simplified Program Logic for Multiple Interactive Terminals

Changing A Terminal File's Characteristics

The file characteristics (e.g., line length or record size, detabbing, device type, operational mode) of a terminal are established at the time of system configuration. These characteristics can be changed by the file system user at execution time, before the file associated with the device is opened, through use of the Set Terminal Characteristics command (STTY) or macro call ($STTY).

Of particular interest to the communications user are the STTY arguments that control the operational modes of a device. Examples of operational modes include echoplex, use of control bytes, and optional end-of-message processing. The user can specify operational modes by specifying a -MODES argument or by setting bits of a device specific word.

SPECIFICATION BY -MODES ARGUMENT

The file system user can most conveniently specify operational modes by means of the -MODES arguments of the STTY command. For example, to specify the terminal's echoplex feature, the user enters -MODES ECHO. Conversely, the user enters -MODES ^ECHO to suppress the echoplex feature. To reset all operational modes to those designated at the time of configuration, the user invokes the control argument -RESET.

SPECIFICATION BY DSW BIT SETTINGS

In some instances, the file system user may be required to specify the operational modes of a device by setting bits in the device-specific word (DSW) I_DVS in the IORB. This requirement occurs when the user wishes to alter an operational mode for which a -MODES argument has not been defined.

Specification by DSW bit settings is accomplished through the DSW1 and DSW2 arguments of the STTY command or $STTY macro call. The DSW1 argument is used to change the I_DVS field in connect and disconnect IORBs that the file manager issues against a communications device; DSW2 is used to change the I_DVS field in the read and write IORBs that the file manager issues against the same communications device. A user, for example, can specify BSC 2780/3780 control byte processing by setting bit 4 in DSW1 to zero.

To change a terminal's operating characteristics through the bit settings of the DSW, proceed as follows.

1. Determine which line protocol handler is servicing the terminal to be modified. One source for this information is the system's Configuration Load Manager (CLM) file (usually >SID>CLM_USER). In this file, a DEVICE directive names each device supported by the file system; each DEVICE directive in the file is paired with a station-defining directive that specifies the LPH serving the device.

2. Ascertain the operational characteristics established for the device at the time of configuration. The operational characteristics of a device are determined by the device-specific words of an IORB. The bit values of the device-specific words are set by the system; these default values are shown in Table 4-6 below. The user should consult the appropriate sections in this manual for the significance of particular bits in device-specific words. The sections that should be referenced are as follows:

Device_Unit (LPH)	Section
Asynchronous Terminal Driver (ATD)	8
Synchronous Terminal Driver (STD)	9
Polled VIP Emulator (PVE)	10
BSC Line Protocol Handler (BSC)	11
TTY Line Protocol Handler (TTY)	12

The system-defined default values for device-specific words can be changed at the time of configuration by means of the STTY directive.

3. To change temporarily a DSW value that is in effect, enter a new value by means of the STTY command or $STTY function. The new value will remain in effect only during the current session. To permanently change the operating characteristics of a device, use the STTY directive (described in the System Building and Administration manual).

Table 4-6. System Defaults for DSW1 and DSW2

Device_Unit	DSW1	DSW2
TTY	0000	0030
BSC	0000	0000
PVE	0000	0000
XBSC	0040	0000
ATD	0000	0030
STD	0103	0010

COMMUNICATIONS PROCESSING THROUGH PHYSICAL I/O

The physical input/output (I/O) interface permits direct control by the user over communications processing. Used only with Assembly language programs, the physical I/O interface enables communications applications to:

- Call appropriate line protocol handlers (LPHs) directly through the communications subsystem rather than through the file system.
- Control the data structure, specifically the input/output request block (IORB), that directly affects device operations and/or characteristics.

Physical I/O

The following conventions apply to use of physical I/O:

- The I/O request block (IORB) is the standard control structure used by an LPH.
- An application program requests an I/O transfer by issuing a Request I/O ($RQIO) macro call.
- When configured, all LPHs and associated devices are identified by a set of unique LRNs at the time of system building. A line protocol handler is invoked when its LRN is included in the IORB for a subsequent $RQIO macro call.
- At the time of the $RQIO macro call, the B4 register contains the address of the IORB supplied by the application program.
- Bit F of IORB field I_CT1 must be set to 1; this is required for any I/O request.
- Before giving up control, the LPH maps the hardware return status into the status word I_ST of the application's IORB.

Table 4-7 lists the status codes that are returned (in the left byte of I_CT1) to indicate the result of an I/O request.

Table 4-7. I/O Request Status Codes Returned in I_CTl

Code Number (Hexadecimal)	Meaning
0	No error, operation complete
1	Request block already busy (T=1)
2	Invalid LRN
3	Illegal wait
4	Invalid field values in the IORB
5	Device not ready
6	Device timeout on other than connect
7	Hardware error
8	Device disabled
9	File mark encountered
A	Controller unavailable
B	Device unavailable
C	Inconsistent request
F	EOT received (for BSC3780 and ATD stream mode)
10	Device timeout on connect

NOTES

1. The 08 (device disabled) status is returned on an I/O request when the application has disabled the logical resource. It is also returned if a connect or disconnect has been issued against a line or device that is currently being connected (by a prior connect order) or disconnected (by a prior disconnect order).

2. The 0B (device unavailable) status is returned with every read or write IORB that has been aborted by a disconnect request with queue abort. This status can also indicate the loss (drop) of a communication line.

Table 4-7 (cont). I/O Request Status Codes Returned in I_CT1

3. When the 07 (hardware error) status is found in I_CT1 or in $R1 on a resume after wait, look at the IORB field I_ST to identify the specific error.
4. The 0C (inconsistent request) status indicates illogical I/O requests: read or write before connect, duplicate connect or disconnect requests, write after disconnect.

Using Physical I/O

Two fields within the IORB specify the operation to be performed.

1. The function code (Table 4-10), indicated by bits C through F of I_CT2 in the IORB (Table 4-8), specifies the particular operation.

2. The I_DVS item in the IORB, used with the function code, specializes the input/output order.

To request execution of an I/O operation, the application, with the $RQIO macro call, must transfer control to the physical I/O interface. At the time of the request, the B4 register must contain the address of the IORB being requested. The $RQIO macro routine initiates the I/O operation, and returns control to the requesting application.

The IORB may specify either synchronous or asynchronous execution.

When the IORB specifies synchronous I/O (bit 9 of I_CT1=0), return to the calling application is delayed by the Executive until the I/O operation is complete. On return of control to the application, both the return status field in I_CT1 of the IORB and the R1 register will contain one of the status codes shown in Table 4-7.

When the IORB specifies asynchronous I/O (bit 9 of I_CT1=1), control returns immediately without waiting for I/O completion, and the instruction at the return point is executed as soon as the system initiates the requested I/O operation.

To obtain the completion status (in R1 register) when using asynchronous I/O, the application should issue a $WAIT or $TEST macro call. The $WAIT macro call blocks execution of the application until the requested I/O operation is marked as complete. At completion of the I/O operation, the application should first check the R1 register to see that the I/O request was successful. Any error will be defined there. Hardware errors will be indicated in the IORB software status word I_ST

(see Table 4-9). The $TEST macro call returns the completion status of the IORB if the I/O transfer has completed, or returns status 0801 if I/O has not completed. The $TEST macro call allows the application to continue processing pending completion of an I/O transfer, whereas $WAIT does not.

Residual range, indicated in the IORB, shows how much of the requested data was transferred. The residual range value in I_RSR of the IORB is meaningful only when the A-bit in the I_ST item (Table 4-8) of the IORB has been set on.

DATA STRUCTURES

Data structures control the interactions among an application program, its line protocol handlers, and the devices it uses. The input/output request block (IORB) is the interface between the application and line protocol handler. The IORB and its use are described below in general terms. Later sections describe the contents of specialized IORBs for each of the line protocol handlers.

Input/Output Request Blocks

The IORB is the standard means for requesting a physical I/O service. As described in this section, the IORB is used with physical I/O communications interfaces. The physical I/O part (through 13+2*$AF in Figure 4-2) is directly usable at the physical I/O interface. The logical part (beginning with 14+2*$AF) is used by forms processing software, by the local mail facility (interprocess communication), and by the message group request blocks MGIRB, MGCRB, and MGRRB.

Generated by the Input/Output Request Block macro call ($IORB), the IORB contains all the information that an application requesting an I/O service must specify to define the operation to be performed. Specifically, the IORB includes the following:

- Logical resource number (LRN) that identifies the I/O device being addressed
- Location and size of the buffer to be used for physical I/O transfers
- Type of operation as specified by the function code and optional device-specific word
- Information, concerning results of the I/O request, returned by the line protocol handler to the application after I/O completion.

When the IORB is used with a $RQIO macro call, the device named in the IORB should have been previously reserved by a Get File ($GTFIL) macro call. The logical resource number (LRN) required by the IORB can be obtained by issuing a Get File Information ($GIFIL) macro call. For further details, see the description of the Request I/O ($RQIO) macro call in Volume II.

Figure 4-2 shows the format of the IORB. Table 4-8 defines the separate entries in the IORB. Later sections in the manual describe the significance of the device-specific word (I_DVS), software status word I_ST, and other IORB words for the various line protocol handlers.

NOTES

1. The labels used in the figure to identify IORB fields (e.g., I_CTl, I_ADR) can be generated by the $IORBD macro call, described in Volume II.

2. The offset symbol $AF signifies the number of words required to specify a memory address. In this system, $AF is equivalent to two words.

3. The asterisk (*) in the formulas in the "Word" column of Figure 4-2 and Table 4-8 is a multiplication sign.

4. The shaded fields in figure 4-2 are for system use only. Fields not shaded must be initialized by the application requesting the I/O operation.

IORB SOFTWARE STATUS WORD (I_ST)

The line protocol handler maps into the IORB software status word I_ST (Table 4-9) the return status of the hardware or line protocol handler.

The bit settings in the software status word I_ST indicate to the application the status of the hardware, as shown in Table 4-9.

The meanings of bit settings in the software status word I_ST for specific devices are shown in tables in later sections that describe the line protocol handlers for those devices.

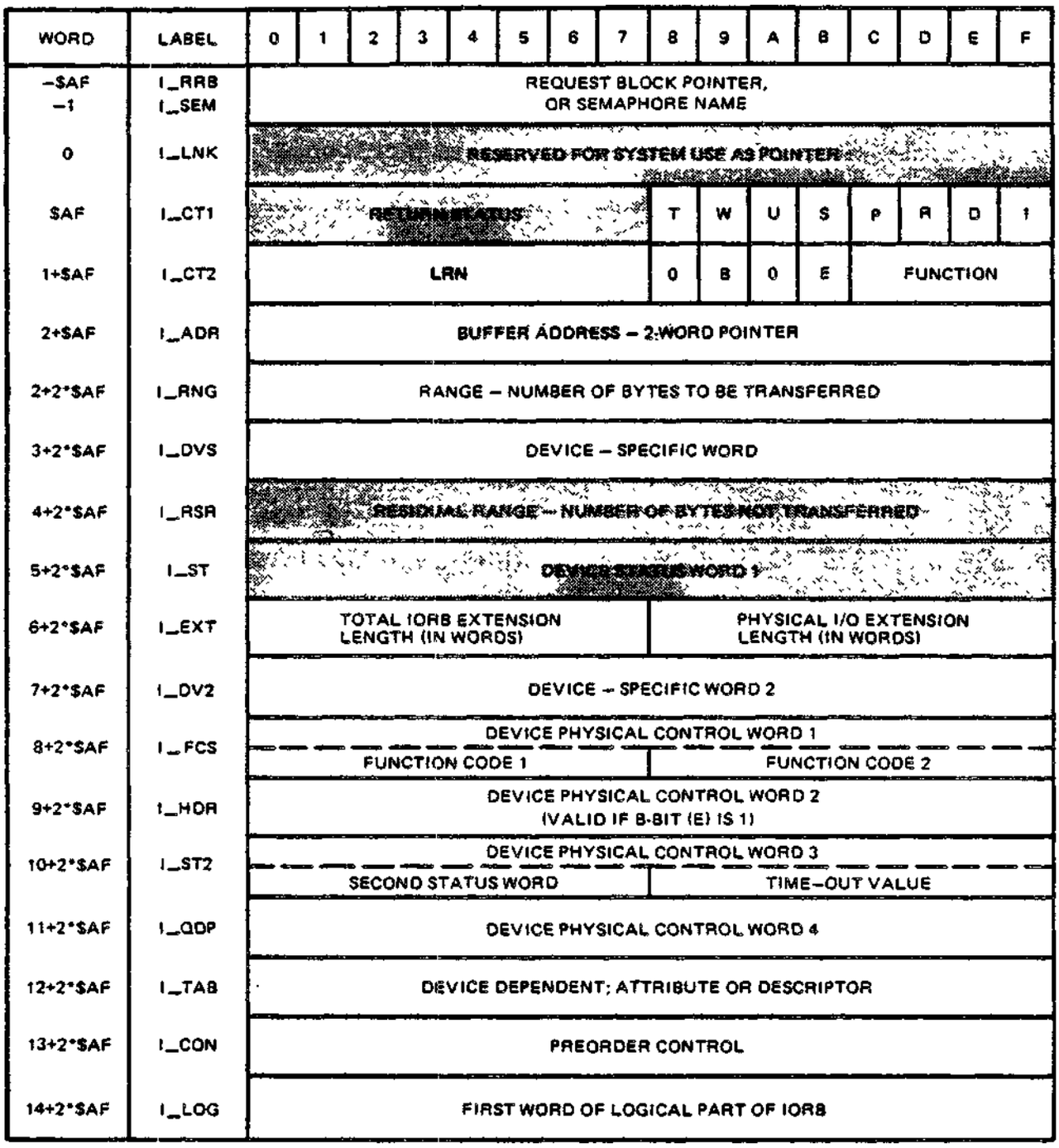


Figure 4-2. Communications Input/Output Request Block (IORB)

Table 4-8. Communications Input/Output Request Block (IORB)

Word	Label	Bits	Description
-$AF	I_RRB/ I_SEM	0-31	Depending on the S- or R-bits of I_CT1, this word contains a task request block pointer (R-bit on) or a semaphore name (S-bit on). Set by user; used by system at termination of request.
0	I_LNK	0-31	Reserved for system use; two-word pointer.
$AF	I_CT1	0-7	Return status. (See Table 4-7).
		8 (T)	This bit is set (on) while the request using this IORB is executing; it is reset when the request terminates. The system controls this bit; user should not change it.
		9 (W)	Wait bit. Set by user when the requesting task is not to be suspended pending the completion of the request that uses this IORB. If W = 0, then the D, R, and S bits may not be set.
		A (U)	User bit. User may or may not use this bit; system does not change it.
		B (S)	Release semaphore indicator. Values: 0 = No semaphore in I_SEM. 1 = Release, on completion, semaphore item named in I_SEM.
		C(P)	Must be set by user if IORB is to be referenced by a Wait Any ($WAITA) macro call. If set, IORB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D (R)	Return IORB indicator. Values: 0 = No request pointer in I_RRB. 1 = Dispatch task request block named in I_RRB; after completion of this request, the system executes $RQTSK, using I_RRB.

Table 4-8 (cont). Communications Input/Output Request Block (IORB)

Word	Label	Bits	Description
$AF (cont)	I_CT1 (cont)	E (D)	Delete IORB indicator, used usually with B (S) and D (R) bits. 0 = No delete. 1 = When task terminates, return memory to the pool where IORB is the first entry of its memory block.
		F	I/O bit. Must be set to 1.
1+$AF	I_CT2	0-7	Logical resource number (LRN); identifies device to be used.
		8	Must be 0.
		9 (B)	Byte index. 0 = buffer begins in leftmost byte of word; 1 = buffer begins in rightmost byte.
		A (P)	Reserved for system use.
		B (E)	Extended IORB indicator. 0 = Standard (nonextended) IORB. 1 = IORB extended as specified by I_EXT. Set by user. (See I_EXT below.)
		C-F	Function code. See Table 4-10.
2+$AF	I_ADR	0-31	Buffer address; 2-word pointer.
2+2*$AF	I_RNG	0-15	Range. Indicates number of bytes to be transferred.
3+2*$AF	I_DVS	0-15	Device-specific information. Set by user.
4+2*$AF	I_RSR	0-15	Residual range. Indicates the number of bytes not transferred. Filled in by the system on completion of the order.
5+2*$AF	I_ST	0-15	Status word. Reflects the mapping of the hardware status into software status format. Set by system after I/O completes. Used also by the ATD and STD LPHs as a peripheral address field.

Table 4-8 (cont). Communications Input/Output Request Block (IORB)

Word	Label	Bits	Description
6+2*$AF	I_EXT	0-7	Left byte: Number of words in the IORB extension, not including this I_EXT word.
		8-F	Right byte: Number of words in physical part of IORB extension, not including this I_EXT word; must be less than or equal to total extension length shown in the left byte. This word applies only when the B (E) bit in I_CT2 is 1.
7+2*$AF	I_DV2	0-F	Device-specific word 2. Contains device-specific information.
8+2*$AF	I_FCS	0-F	Device physical control word 1.
9+2*$AF	I_HDR	0-F	Device physical control word 2.
10+2*$AF	I_ST2	0-F	Device physical control word 3.
11+2*$AF	I_QDP	0-F	Device physical control word 4.
12+2*$AF	I_TAB	0-F	Device physical control word 5.
13+2*$AF	I_CON	0-F	Device physical control word 6.
14+2*$AF	I_LOG	0-F	First word of logical part of IORB. Used by forms processing software, in message control, and by local mail message group request blocks.

Table 4-9. Software (I_ST) Status Codes

Bit in IORB's I_ST	Meaning When Bit Set On
0	-
1	Read error (PVE)
2	Data service rate error
3	Lost line bid or RVI received (BSC)
4	Communication control block service error
5	No stop bit on character input (TTY); conversational reply received (BSC3780); IORB purged because of BREAK signal (ATD, TTY)
6	Long record (BSC, ATD)
7	ITB/ETB or ETX received (BSC); poll failure (PVE)
8	Framing error (ATD); NAK limit reached (PVE)
9	Checksum or parity error limit reached (PVE); parity error (ATD)
A	Nonzero residual range
B	Phone disconnect
C	End-of-transmission received (BSC)
D	Transparent message received (BSC)
E	NAK limit reached (BSC)
F	Nonexistent resource; bus parity error; fatal uncorrectable memory error

Communications Function Codes

All line protocol handlers perform similar functions for the devices and applications that they service. These functions are performed by the line protocol handler's request and interrupt processing codes.

An application can request specific functions by providing a function code in the IORB supplied when it requests I/O service. The application uses the last four bits of its IORB's I_CT2 entry (see Figure 4-2) to enter the function code for the functions summarized in Table 4-10.

The connect and disconnect functions may be used with non-communications devices, in which case they are processed as "no-ops". Thus, no matter how connected to the system, all TTY devices and noninteractive (e.g., card reader and printer) devices can be controlled by the same application program. This provision is useful for program development and test purposes.

Table 4-10. Communications LPH Function Codes

Function Code in IORB	Communications Function
1	Write
2	Read
5	Define-form (used only by the ATD LPH)
9	Read break
A	Connect
B	Disconnect

WRITE FUNCTION (CODE 1)

This function allows data to be written to a specific device. When a line protocol handler (LPH) receives a write request, it transfers the indicated data from the application's buffer to the device, according to the information supplied in the device-specific word of the application's IORB.

READ FUNCTION (CODE 2)

This function allows data to be read from a specific device. When the LPH receives a read request, it tranfers data from the device to the application's buffer, according to the information supplied in the device-specific word of the application's IORB.

DEFINE-FORM FUNCTION (CODE 5)

This function is used by the ATD LPH for forms processing to define fields, their subfields, and their attributes. A define-form order does not itself result in actual physical I/O. (Refer to Section 8 for more details.)

READ BREAK (CODE 9)

This function allows an application to be notified of an operator-generated break condition on synchronous or asynchronous terminals. The function also allows for the selective cancellation of outstanding read break orders. (Refer to Section 8 for more details.)

CONNECT FUNCTION (CODE A)

The connect function provides a logical and physical connection between an application program and a communications device.

As a logical function, the connect function is a request to use the specified communications device. If that resource is being used, an error return results. In that case, the application must determine whether that resource is sharable (as established by the installation's procedures) and proceed accordingly.

As a physical function, the connect function establishes a physical path to the communications device associated with the specified logical resource number (LRN). This implies, when the device is to be connected over a switched line, that the system software should complete call establishment on the line associated with that device. The request times out after five minutes.

If the connect function is not completed, the system will not process any requests for the communications device, and will return an error status.

The connect function must be requested before any other function, since communications devices are configured into the system in a disconnected state.

DISCONNECT FUNCTION (CODE B)

The disconnect function provides both the logical (normal and abnormal) and physical disconnection between the application and a communications device.

As a logical function, the disconnect function indicates that the use of the designated device is to be terminated.

For a logical disconnect, issue a disconnect request (function code B) with the E-bit in I_DVS set off (dequeue remaining IORBs for device) and the F-bit in I_DVS set on (do not hang up phone). At this point, any pending read or write requests are returned to the application program with a B status (device unavailable). Continued use of the device requires that the application program issue a connect.

As a physical function, the disconnect function must specify, by setting the F-bit in I_DVS to 0, the physical disconnection of a line.

Section 5
DATA STRUCTURE GENERATION

This section summarizes the macro routines that generate and define system data structures. There are two kinds of data structures: those that apply to system control functions and those that apply to file system functions. The macro calls that generate both kinds of data structures are described in detail in Volume II of this manual. The formats of the generated data structures are tabulated in Appendix D.

SYSTEM CONTROL DATA STRUCTURES

System control data structures that are visible to the user consist of the following:

- Request blocks
- Parameter block and wait lists.

Request Blocks

When requesting certain operations, tasks generate request blocks in order to specify the parameters of the requested operation. The first five words of all request blocks are identical in format; these words pass parameters to the system. The W-bit, for example, in the third word of request blocks, specifies whether or not the requesting task is to be suspended until the requested operation is completed. Additional words convey to the system information specific to the request block type.

One type of request block, the task request block, passes parameters to the requested task as well as to the system. These additional parameters are arguments that control the execution of the task being requested. They are entered into a variable-length field of the task request block called an argument list.

Table 5-1 lists the request blocks and the macro calls that generate them.

Table 5-1. Request Blocks

Request Block	Macro Call
Clock request block (CRB)	$CRB
Input/output request block (IORB)	$IORB
Message group request blocks	
Message group control (MGCRB)	$MGCRB
Message group initialization (MGIRB)	$MGIRB
Message group recovery (MGRRB)	$MGRRB
Semaphore request block (SRB)	$SRB
Task Request block (TRB)	$TRB

The arguments supplied with each of the above macro calls sets values for fields of the corresponding request block. For example, the first argument of the Input/Output Request Block ($IORB) macro call specifies the logical resource number (LRN) of the device to perform the input/output operation. The number specified by this argument is placed in the fifth word of the request block generated by the $IORB macro call.

REQUEST BLOCK OFFSETS MACRO CALLS

Each request block macro call is paired with a request block offsets macro call. Request block offsets macro calls generate tags for every entry in a corresponding request block, allowing symbolic references to request block fields by application code. These tags are not generated by request block macro calls. An application may use a request block macro call to construct a request block, and then issue a request block offsets call to facilitate modification of the existing block by executing code.

Unlike the arguments of request block macro calls, the tags generated by offset macro calls refer to all fields of the corresponding request block. Offset tags refer to fields in which values are returned by the system, whereas macro call arguments refer only to fields in which values are entered by the user.

As mentioned above, the first five words of all request blocks are identical. Each offset macro call, however, refers to these words by different tags. The fourth word of the semaphore request block, for example is S_CTl, whereas the fourth word of the task request block is labeled T_CTl. The programmer, therefore, can include several types of offset macro calls in an application without multiply defining symbols.

No arguments are specified with offsets macro calls. Only one offsets macro call of a particular type is allowed in an application.

Macro calls that generate offsets tags for request blocks are listed below:

Clock Request Block Offsets	$CRBD
Input/Output Request Block Offsets	$IORBD
Message Group Control Request Block Offsets	$MGCRT
Message Group Initialization Request Block Offsets	$MGIRT
Message Group Recovery Request Block Offsets	$MGRRT
Semaphore Request Block Offsets	$SRBD
Task Request Block Offsets	$TRBD

Parameter Block and Wait Lists

The parameter block and wait lists are system control data structures that differ in format from request blocks.

A parameter block is equivalent to the task request block's argument list, mentioned above; it is generated by the Parameter Block ($PRBLK) macro call. Parameter blocks are a standard means of passing arguments between tasks. By specifying the number and length of arguments, as well as the arguments themselves, a parameter block allows the receiving task to locate each argument in the list (or block).

A wait list is a list of request blocks to be serviced before the task issuing the wait list macro call completes its own execution. A wait list consists of a count of the number of request blocks to be waited on, followed by the request blocks' addresses. The list is generated by the Wait List ($WLIST) macro call. Another macro call, Wait on Request List ($WAITL) causes the task manager to scan the wait list and activate the waiting task when any of the listed requests are marked as completed.

A multiple wait list contains the same information as does the wait list; in addition, it specifies the number of request blocks that must be completed before a waiting task is to be activated. A multiple wait list is generated by the Generate Multiple Wait List ($WLSTM) macro call.

FILE SYSTEM DATA STRUCTURES

A file information block (FIB) is used by running applications to request input/output operations. Other data structures are used outside of program execution by functions that create and modify files, or return information about files already created. Both types of data structures are discussed below.

File Information Block

The file information block is the means by which an application passes to the file system the parameters of a requested input/output operation. The fields of the FIB specify such items as a file's logical resource number (LFN), by which the system identifies the file; the record or block size; and the address of the user's buffer.

The following macro calls use an FIB:

Open File	$OPFIL
Close File	$CLFIL
Test File	$TIFIL, $TOFIL
Read Record	$RDREC
Write Record	$WRREC
Rewrite Record	$RWREC
Delete Record	$DLREC
Read Block	$RDBLK
Write Block	$WRBLK
Wait Block	$WTBLK

FILE INFORMATION BLOCK MACRO CALL

The file information block is generated by the File Information Block ($FIB) macro call. An $FIB macro call can do one of the following:

- Build a new FIB with default values determined by the system
- Build a new FIB, specifying its contents by means of arguments supplied with the call
- Generate instructions to alter the contents of an existing FIB.

As explained in Section 3, the file system performs three functions: data management, file management, and storage management. An FIB pertinent to one type of function may not be pertinent to another type. Data management involves the transfer of logical records; storage management, the transfer of blocks of records. The fields of an FIB applicable to data management, therefore, would specify the size and location of logical records; the fields of an FIB applicable to storage management, the size and location of record blocks. For this reason, the FIB macro call has two sets of arguments, pertaining to data/file management and storage management.

FIB OFFSET MACRO CALLS

For the same reason that the $FIB has more than one set of arguments, there are several macro calls that generate FIB offset tags. (The use of offset tags is explained earlier in this section.) The FIB offsets macro calls are:

```
$FIBDM
$FIBSM
$TFIB
```

The $FIBDM and $FIBSM macro calls generate sets of tags that are specific to data/file management and storage management, respectively. A third offsets macro call, $TFIB, generates two sets of tags, applicable both to data/file and to storage management. The $TFIB macro call would be issued by an application requesting both data/file management and storage management services.

Macro Call Argument Structures

Macro calls that create and modify files, or return information about existing files must specify many parameters, as a file can take many different forms. Typically, these macro calls have a single argument that points to a list of arguments, or an argument structure. Offsets macro calls are available to facilitate modifying or referring to the fields of an argument structure. Table 5-2 lists the file system macro calls that require argument structures and the offsets macro calls that supply tags for these structures.

Table 5-2. Argument Structures and Offsets Tags

Calls Requiring Argument Structures	Calls Generating Offset Tags
Create File ($CRFIL)	Create File Parameter Block Structure Offsets ($CRPSB)
	Create File Key Descriptor Block Offsets ($CRKDB)
Get Device Information ($GIDEV)	Get Device Information Parameter Structure Block Offsets ($DIPSB)
Get file Access Rights ($GAFIL)	Get File Access Rights Parameter Structure Block Offsets ($GAPSB)
Get File Information ($GIFIL)	Get File Information Key Descriptor Block Offsets ($GIKDB)
	Get File Information Parameter Structure Block Offsets ($GIPSB)
	Get File Information File Attribute Block Offsets ($GIFAB)
Grow File ($GRFIL)	Grow File Parameter Structure Block Offsets ($GRPSB)
Modify File ($MDFIL)	Modify FIle Parameter Structure Block ($MDPSB)
Shrink File ($SHFIL)	Shrink File Parameter Structure Block Offsets ($SHFIL)

Size Tags

Data structures for file system macro calls can either be declared statically or built dynamically. In the latter case, memory for the structure is dynamically obtained by means of the Get Memory ($GMEM) macro call at the time of execution. The memory thus obtained should be cleared to zeros to ensure that fields of the structure reserved for future are zero-filled. Each offset macro call generates a size tag for specifying the size of the corresponding data structure. The size tag can be used to specify the amount of memory requested (when issuing the Get Memory macro call), or used to clear the structure to zeros.

Example:

$B4 points to a file information block (FIB). The structure is cleared with the instructions:

```
        LDV   $R1,F_SZ-1        R1=SIZE OF FIB MINUS 1
    $A  CL    $B4.$R1           CLEAR ONE WORD
        BDEC  $R1,>-$A          LOOP UNTIL ALL WORDS CLEARED
```

Section 6
DEVICE DRIVERS

This section describes the internal system software known as device drivers and some related data structures, principally the input/output request block (IORB), by which the device driver is controlled. A device driver performs all data transfers between a peripheral device and an application program requesting input/output. (A peripheral device is one attached to a multiple device controller (MDC)). Line protocol handlers analogously perform input/output between applications and communications devices, which are attached to a multi-line communications processor (MLCP). The remainder of this section describes peripheral device drivers. Line protocol handlers are described in later sections.

INPUT/OUTPUT DRIVERS

Applications can request and instruct drivers directly by means of the Request Input/Output ($RQIO) and Input/output Request Block ($IORB) macro calls. Applications invoke drivers indirectly when issuing file system macro calls such as Read Record ($RDREC) and Write Record ($WRREC). When executing these calls, the file system generates IORBs to instruct the drivers.

Drivers are reentrant programs capable of supporting the concurrent operation of several devices at the same time. The priority level at which they run is selected by the user when the system is configured. Requests by applications for I/O activate the drivers, which in turn initiate data transfer that is simultaneous with the operation of the central processor. Drivers generate an interrupt to the central processor when the transfer of data is terminated.

DEVICE DRIVER DATA STRUCTURES

Two data structures control the interaction between an application program, its device drivers, and the devices the program uses. These structures are the input/output request block (IORB) and the resource control table (RCT).

The IORB is the interface between the application and its device driver. Through the IORB, the application defines the I/O service that it wishes to be performed. Also, the IORB contains information returned by the driver to the requesting task concerning the outcome of the I/O request.

The resource control table (RCT) is the interface between the driver and its device(s), and is not normally accessible to users of Honeywell-supplied drivers described in this section. The RCT is used by those who write their own device drivers; it is described in the System Building and Administration manual.

DEVICE DRIVER CONVENTIONS

The following conventions apply to all input/output device drivers.

- The I/O request block (IORB) is the standard control structure used by a driver. It is described later in this section.
- The $RQIO macro call is used to request a driver.
- The B4 register contains the address of the IORB supplied by the caller; the IORB contains the LRN of the device to be used.
- The I/O-specific words of the IORB (I_CT2 through I_DVS) are not modified by the driver.
- If a device becomes inoperable, it can be disabled with an operator command and another device can be substituted.
- Drivers are reentrant and interrupt driven; one driver supports many devices of the same type.
- Synchronous and asynchronous I/O are supported.

- The hardware status is always mapped into the software status word in the task's IORB (I_ST) before the driver relinquishes control.

Driver Functions and Function Codes

All drivers perform similar functions on behalf of the devices and application tasks they service. These functions are carried out by the driver's request processing and interrupt processing code.

The application task requests specific functions by providing a function code in the IORB that it supplies when it requests I/O service. These specific function codes are summarized in Table 6-1 and discussed under the specific function heading in the following pages.

The application task uses the last four bits of the IORB entry I_CT2 to enter the function code for the functions summarized in Table 6-1.

CONNECT FUNCTION (fc=A)

This function provides the logical and physical connection between an application program and an interactive peripheral device (i.e., a KSR/ASR device connected to a multiple device controller). The function may also be used with noninteractive devices for program compatibility. The driver of a noninteractive device treats this function as a NOP and immediately posts the IORB back to the requester with successful status (operation complete).

DISCONNECT FUNCTION (fc=B)

This function code provides the logical (normal and abnormal) and physical disconnect between an application program and an interactive device.

The disconnect function as a logical function indicates that use of the indicated device is terminated. Termination may be either normal or an abort of all queued read or write requests issued by this user program.

WAIT ONLINE FUNCTION (fc=0)

The "wait online" function allows a caller to wait until a device becomes ready for use, or until a specific time interval has passed.

All noncommunications devices (except KSR-like devices) generate interrupts when their availability changes. For example, when a printer runs out of paper, an interrupt is generated and the device is not ready for use; when the paper is installed and the device is again ready, another interrupt is generated.

When a driver receives a service request from a task using the "wait online" function code in the IORB that it supplies (0000 in the last four bits of I_CT2), and the device is not ready, the driver sets a timer for 5 minutes and suspends. When the driver is reactivated, either by a ready interrupt from the device or by a timeout, it deactivates the timer, checks the device-ready bit in the hardware status word, and places a 0 or 6 value in the return status field of the IORB depending on the condition of that bit. See Table 6-2 and the return status codes for the $RQIO macro call (which is described in Volume II). The rightmost 2 digits of the 4-digit hexadecimal status code are placed in the return status field.

The wait online function should not be issued to a device that is currently ready for use unless you expect it to become unavailable for a limited time (e.g., the operator has been instructed to change a volume mounted on a disk device currently in use).

Table 6-1. Input/Output Function Code

IORB Function Code	ASR/KSR Keyboard Printer	Card Reader	Card Reader/Punch	Device		Magnetic Tape
				Printer	Disk	
0	Wait Online	Wait Online	Wait Online	Wait Online	Wait Online	Wait Online
1	Write	NA	Write (Punch)	Write	Write	Write
2	Read	Read	Read	NA	Read	Read
3	NA	NA	Write File mark (Punch)	NA	NA	Write file Mark
4	NA	NA	NA	NA	NA	Position Block*
5	NA	NA	NA	NA	Format write	NA
6	NA	NA	NA	NA	Format Read	Position File**
9	Break Notification	NA	NA	NA	NA	NA
A	Connect	Connect	Connect	Connect	Connect	Connect
B	Disconnect	Disconnect	Disconnect	Disconnect	Disconnect	Disconnect
E	NA	NA	NA	NA	Read Disabled Device	Read Disabled Device

NA = Not Applicable

* Positive range of one is forward space to start of next block.
Negative range of one is backspace to beginning of previous block.

** Positive range of one is forward space to next tape mark.
Zero range is backspace to previous tape mark.
Negative range of -1 is rewind to BOT.
Negative range of -2 is rewind to BOT and unload.
Negative range of -3 is write at EOF gap.

Table 6-2. Return Status Codes (Last Two Digits)

Code Number (Hexadecimal)	Meaning
00	No error, operation complete
01	Request block already busy (T=1)
02	Invalid LRN
03	Invalid wait
04	Invalid parameters
05	Device not ready
06	Device timeout on other than connect
07	Hardware error
08	Device disabled
09	File mark encountered
0A	Controller unavailable
0B	Device unavailable
0C	Inconsistent request
10	Device timeout on connect
11	Write protect error
17	Memory access violation

NOTES

1. When status 07 is returned, look in I_ST to identify the specific hardware error.

2. Status 0B is returned with every read or write IORB that has been aborted by a disconnect request with queue abort. The disks and tapes are disabled until the system's automatic volume recognition routine calls the enable device function.

3. Status 0C indicates illogical peripheral driver requests (e.g., read or write before connect; duplicate connect or disconnect requests; write after disconnect).

WRITE FUNCTION (fc=1)

The write function is available for all devices except the card reader. This function allows the writing of data to a particular device. When a driver receives a write request, it transfers the indicated data from a user buffer to the device according to the specifications supplied in the task's IORB.

READ FUNCTION (fc=2)

The read function is available for all devices except local and remote printers. This function allows reading data from a particular device. When a driver receives a read request, it transfers the data from the specified device to a user buffer according to the specifications supplied in the requesting task's IORB.

READ DISABLED DEVICE FUNCTION (fc=E)

This function, available only to disk or magnetic tape devices, allows the driver to bypass the device-disabled test during validity checking.

This function is used by the system's automatic volume recognition (AVR) module, which recognizes the volume label of the volume on the disabled device, then enables the device so that attempts to read data from it can continue.

WRITE TAPE MARK FUNCTION (fc=3)

The write tape mark function, which is available to magnetic tape devices, allows you to put a mark block on a referenced magnetic tape.

POSITION BLOCK FUNCTION (fc=4)

The position block function, which is available to magnetic tape devices, allows you to position a referenced magnetic tape forward or backward one block.

FORMAT WRITE (fc=5)

The format write function, available only to disk devices, allows you to format a disk device. The number of sectors per track depends upon the device type.

FORMAT READ (fc=6)

The format read function, available only to disk devices, allows you to read all identifier and data fields on a track. The read begins at the first sector following the index mark and proceeds in the order in which the identifiers are recorded.

POSITION TAPE MARK FUNCTION (fc=6)

The position tape mark function, which is available to magnetic tape devices, allows the user to:

- Position forward a referenced magnetic tape beyond the next tape mark
- Position backward a referenced magnetic tape before the current tape mark

- Rewind to BOT
- Rewind to BOT and unload

BREAK NOTIFICATION FUNCTION (fc=9)

This function, available for any terminal device, is a request to notify the issuing task when a break occurs on a specific device. When a break does occur, the driver posts the break notification request and declares the device to be in break mode for the issuing task.

In break mode, all I/O requests issued from the "broken" task are rejected (i.e., posted without any data transfers being started). Execution of a subsequent break notification request will cause the driver to return to normal mode.

INPUT/OUTPUT REQUEST BLOCK

The input/output request block (IORB) contains all information that a task requesting an I/O service can specify to define the operation to be performed. In addition, it contains information returned by the driver to the requesting task concerning the outcome of its I/O request.

Figure 6-1 shows the format of a nonextended IORB. Unshaded fields must be initialized by the task requesting the I/O operation. The shaded fields are set by the driver to return information about the I/O request to the caller, or are controlled by the Executive.

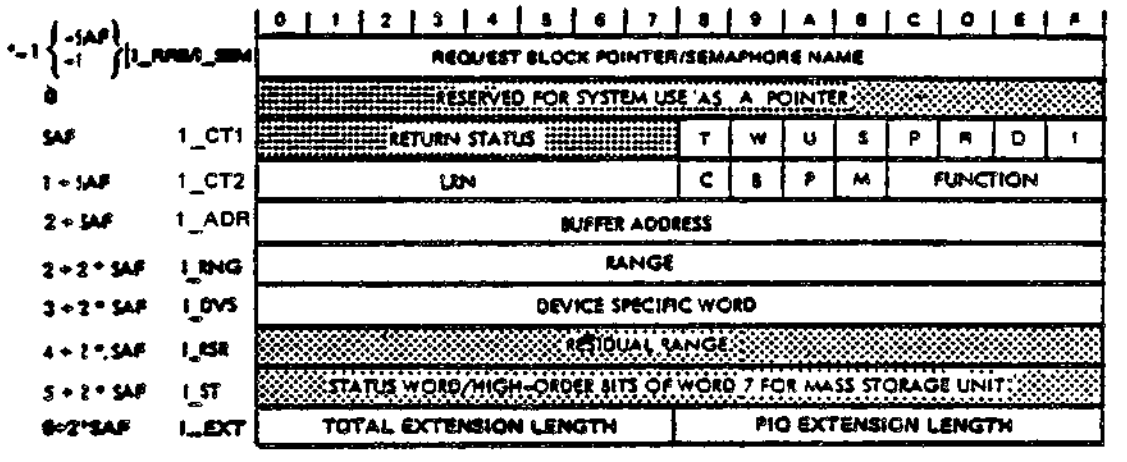


Figure 6-1. Format of I/O Request Block

Table 6-3 defines the specific IORB entries in a nonextended IORB. (See the "Communications Processing Functions" section for descriptions of IORB extensions.) Table 6-4 defines the software status word (I_ST) in the IORB. Device-specific IORB information is provided in the separate device driver descriptions later in this section.

NOTE

The offset labels used to refer to IORB fields (e.g., I_CTl, I_ADR) can be generated by the $IORBD macro call, which is described in Volume II.

Table 6-3. Contents of I/O Request Block

Word	Label	Bits	Contents
-$AF -1	I_RRB/ I_SEM	0-31 0-15	Depending on the S- or R-bits of I_CTl, this word contains a task request block pointer (R-bit on), or a semaphore name (S-bit on); set by user, used by system at termination of request.
0	I_LNK	0-31	Reserved for system use. 2-word pointer to indirect request block.
$AF	I_CTl	0-7	Return status.
		8 (T)	This bit is set (on) while the request using this block is executing; it is reset when the request terminates. System controls this bit; do not change it.
		9 (W)	Wait bit. Set if the requesting task is not to be suspended pending completion of the request that uses this IORB. If W = 0, then the D-, R-, and S- bits may not be set.
		A (U)	User bit. User may or may not use this bit; system does not change it.
		B (S)	Release semaphore indicator.

Table 6-3 (cont). Contents of I/O Request Block

Word	Label	Bits	Contents
$AF (cont)	I_CT1 (cont)		0 = No semaphore in I_SEM, 1 = Release, on completion, semaphore item named in I_SEM.
		C	Must be set by user if IORB is to be referenced by a Wait Any ($WAITA) macro call. If set, IORB can be referenced only by a $WAIT or $WAITA issued by the requesting task.
		D (R)	Return IORB indicator. 0 = No request pointer in I_RRB. 1 = Dispatch task request block named in I_RRB after request timeout. If 1, system executes $RQTSK, using I_RRB, when the task terminates.
		E (D)	Delete IORB indicator, used usually with the B(S) and D(R) bits. 0 = No delete. 1 = Delete and, when task terminates, return memory to pool where IORB is first entry of its memory block.
		F	Implicit task start address. Must always be 1 for IORB.
1+$AF	I_CT2	0-7	Logical resource number (LRN); identifies device to be used.
		8 (IBM)	IBM-type request. Changes interpretation of I_DVS to task word, and I_RSR and I_ST to configuration words A and B, respectively.
		9 (B)	Byte index; 0 = buffer begins in leftmost byte of word. 1 = buffer begins in rightmost byte.
		A (P)	Reserved for system use.
		B (E)	Extended IORB indicator. 0 = Standard (nonextended) IORB. 1 = IORB extended to at least 6+2*$AF items. Set by user. (See I_EXT below.)

Table 6-3 (cont). Contents of I/O Request Block

Word	Label	Bits	Contents
1+$AF (cont)	I_CT2 (cont)	B (E)	Function code. Driver function; see Table 6-1.
2+$AF	I_ADR	0-31	This field contains a 2-word buffer address or, for break notification requests, the ID of the requesting task.
2+2*$AF	I_RNG	0-15	Range. Number of bytes to be transferred.
3+2*$AF	I_DVS	0-15	Device-specific information.
4+2*$AF	I_RSR	0-15	Residual range. Indicates the number of bytes not transferred. Filled in by the system on completion of the order. Used by cartridge disk, Lark disk, and mass storage unit driver as a data offset value on input.
5+2*$AF	I_ST	0-15	Modified device status; shows mapping of hardware status into software status format. See Table 6-4. Set by user as input field high order bits of sector number mass storage unit. Set by system after I/O completion.
6+2*$AF	I_EXT	0-7 8-15	Left byte: Number of words in the IORB extension, not including this I_EXT word. Right byte: Number of words in physical I/O part of IORB extension, not including this I EXT word. This count must be less than or equal to the total extension length specified in the left byte (0-7). This word is present only when the B (E) bit in I_CT2 is 1. (See the "Communications Processing Functions" section for descriptions of IORB extensions.)

Table 6-4. IORB Software Status Word (I_ST)

Bit Position	KSR	Card Reader	Card Reader/ Punch	Printer	Diskette	Lark Disk Cartridge Disk	Cartridge Module Disk and Disk Storage Unit	Magnetic Tape
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0			
2	Over/ underrun	Over/ underrun	Data service rate error	0	Over/ underrun	Over/ underrun	Over/ underrun	Retryable error
3	Even parity error	Mark sense mode	Invalid ASCII code	End of form	Deleted field	Write protect error	Write protect error	Write protect error
4	0	40-column mode	Punch echo or read registration	0	Read error	Read error	Read error	Corrected media error
5	No stop bit	51-column mode	Light/ dark check	0	Device fault	Invalid seek	Invalid seek	Tape mark
6	Long record	External clock track	Card jam	0	Missed data synchronization	Missed data synchronization	Missed data synchronization	BOT
7	Checksum error	Read check	0	0	Unsuccessful search	Unsuccessful search	Unsuccessful search	EOT
8	CC2 termination	ASCII code error	0	0	Two-sided	Missed clock pulse	Missed clock pulse	Long record
9	CC3 termination	0	0	0	0	Missed sector pulse	Successful retry	Nonretryable error
A	0	0	0	0	Seek error	Seek error		0
B	0	0	0	0	0	0	0	Operation check
C	0	0	0	0	0	0	0	High density
D	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0
F	Fatal error	Fatal error	Fatal error	Fatal error	Fatal error	Fatal error	Fatal error	Fatal error

NOTES

1. Nonexistent resource, bus parity, and uncorrected memory errors are combined into bit 15 of I_ST, but each occurrence is noted separatrely in the RCT.
2. The online drivers will flag corrected memory errors and driver or hardware corrected errors in the RCT.
3. This table applies to MDC connected devices only.

CALLER INTERFACE WITH DEVICE DRIVER

To request execution of an I/O operation, the caller must issue a $RQIO macro call with $B4 pointing to the IORB to be serviced. If the IORB specifies synchronous I/O (W-bit reset), the issuing task is suspended until the I/O operation is complete.

If the IORB specifies asynchronous I/O, the instruction at the return point is executed as soon as the system queues the IORB on the driver's level. The application may issue a $WAIT or $TEST macro call when appropriate for the asynchronous request.

Upon return from a synchronous request, the caller must check the R1 register to see if the request was successful. Upon return from an asynchronous request, the caller must check R1 to see if the request was accepted and successfully initiated. For either type of request, any invalid user argument is indicated in R1. Hardware errors are defined in IORB entry I_ST (see Table 6-4).

Residual range denotes how much of the requested data transfer was actually performed. If I_RSR equals zero, all data was transferred. For an asynchronous request, register R1 would be checked on return from the Request I/O macro call; R1, I_ST, and I_RSR should be checked after return from a $WAIT macro call.

Those fields not shaded in Figure 6-1 must be initialized by the task requesting the I/O operation. The remaining fields are set by the driver to return information about the I/O request to the caller or are controlled by the Executive. Table 6-3 describes the purpose of each field.

Other information needed to perform the I/O request is found in the IORB. The caller-supplied standard function code in I_CT2 is mapped by each driver into one or more device functions required to perform the actual request.

The LRN supplied by the caller in the IORB serves as a device identifier.

DEVICE DRIVERS

The remainder of this section discusses the device drivers in the following order:

- Card reader/Card reader-punch driver
- Printer driver
- Disk driver
- ASR/KSR and console drivers
- Magnetic tape driver.

Card Reader/Card Reader-Punch Driver

The card reader and card reader-punch devices are serviced by a single driver. The driver uses six function codes; i.e., read, write, write file mark (reader/punch only), connect, disconnect, and wait online. In addition, its IORB word I_DVS can be coded to define the character code of the input; namely, ASCII or verbatim. These values are specified in the IORB as defined in Table 6-5.

The translation/mapping of these codes from punched card format into memory on reading is described below.

In addition to the standard driver functionality discussed earlier, this driver also:

- Detects and discards unsolicited interrupts
- Detects an end-of-file condition and sets the appropriate return status (ASCII GS character in column 1 of any card=EOF)
- Detects "device not ready" condition and sets appropriate error condition.

ASCII MODE

In this mode, punched cards are processed as shown in Figure 6-2. Each card column consisting of a 12-bit ASCII card code is converted into an 8-bit ASCII byte and stored in the main memory.

The ASCII card code table as specified in American National Standard X3.26 is given in Table 6-5. Note that no multiple punches in rows 1 through 7 are allowed and, thus, the 12-bit card code allows a maximum of 256 unique codes to be defined.

Translation is done by the card reader attachment that also provides a software-visible IORB status indicator that is set whenever an invalid ASCII card code is detected. This error condition is signaled by a 0107 in the R1 register if any card column read had a hole pattern that was not one of the legal hole patterns given in Table 6-5. The invalid card code causes an ASCII-EO (all 1s) code to be loaded in the main memory.

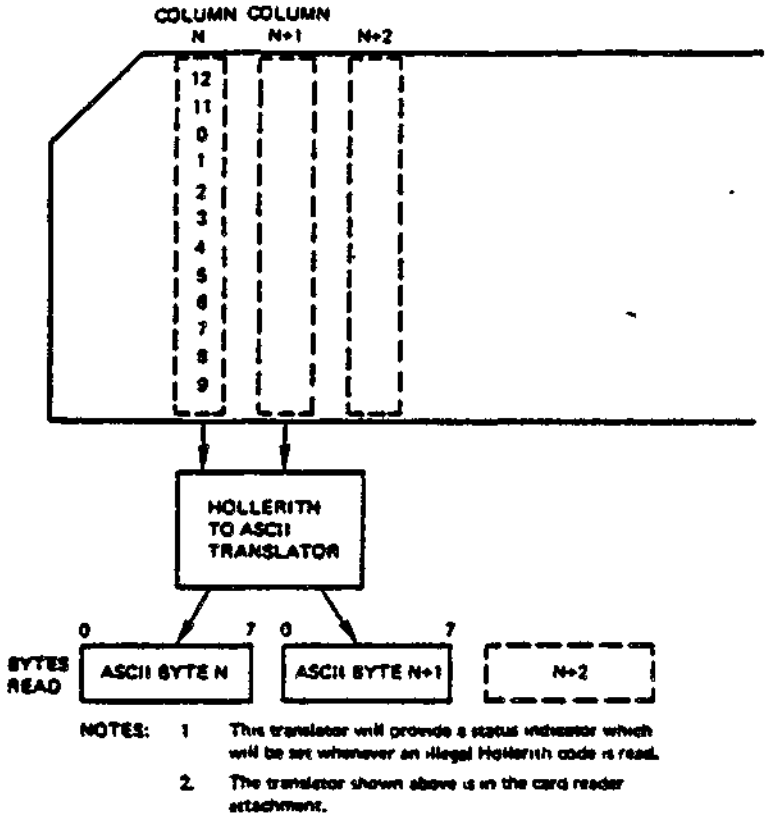


Figure 6-2. ASCII Card-to-Memory Code Formatting

VERBATIM MODE

In this mode, punched cards are processed as shown in Figure 6-3. The card column pattern is stored in bits 4 through 15 of the main memory word with bits 0 through 3 set to zero. All two-hole patterns are valid during a verbatim mode operation. The device-specific fields in the IORB are given below.

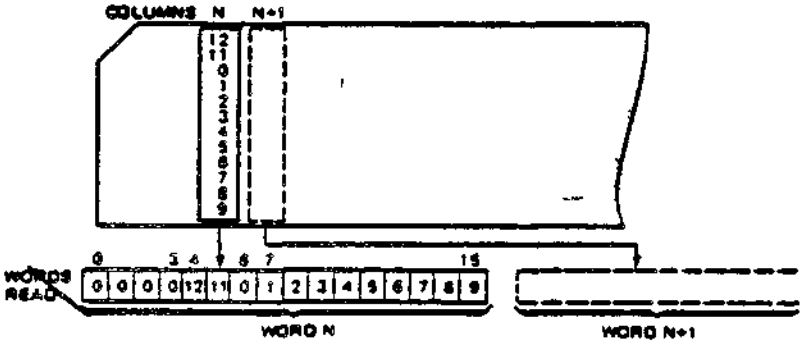


Figure 6-3. Verbatim Mode Formatting

Table 6-5. Hollerith - ASCII Code Table

b6 b5 b4 b3 b2 b1	COL / ROW	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	COL / ROW
0000	0	NUL 12-0-9-8-1	DLE 12-11-9-8-1	SP NO PUNCH	0 0	@ 8-4	P 11-7	` 8-1	p 12-11-7	11-0-9-8-1	12-11-0-9-8-1	12-0-9-1	12-11-9-8	12-11-0-9-6	12-11-0-7	12-11-0-8	12-11-0-8-4	0
0001	1	SOH 12-9-1	DC1 11-9-1	! 12-8-7	1 1	A 12-1	Q 11-8	a 12-0-1	q 12-11-8	0-8-1	9-1	12-0-9-2	11-8-1	12-11-0-9-7	11-0-8-1	12-11-0-9	12-11-0-8-5	1
0010	2	STX 12-9-2	DC2 11-9-2	" 8-7	2 2	B 12-2	R 11-9	b 12-0-2	r 12-11-9	0-8-2	11-0-8-2	12-0-9-3	11-0-9-2	12-11-0-9-8	11-0-8-2	12-11-0-9-2	12-11-9-8-6	2
0011	3	ETX 12-9-3	DC3 11-9-3	# 8-3	3 3	C 12-3	S 0-2	c 12-0-3	s 11-0-2	0-8-3	9-3	12-0-9-4	11-0-8-3	12-0-8-1	11-0-8-5	12-11-0-8-3	12-11-9-8-7	3
0100	4	EOT 9-7	DC4 9-8-4	$ 11-8-3	4 4	D 12-4	T 0-3	d 12-0-4	t 11-0-3	0-9-4	9-4	12-0-9-5	11-0-9-4	12-0-8-2	11-0-8-6	12-11-0-8-4	11-0-9-8-7	4
0101	5	ENQ 0-9-8-5	NAK 9-8-5	% 0-8-4	5 5	E 12-5	U 0-4	e 12-0-5	u 11-0-4	11-9-8	9-5	12-0-9-6	11-0-9-5	12-0-8-3	11-0-8-7	12-11-0-8-5	11-0-9-8-3	5
0110	6	ACK 0-9-8-6	SYN 9-2	& 12	6 6	F 12-6	V 0-5	f 12-0-6	v 11-0-5	12-9-6	9-6	12-0-9-7	11-0-9-6	12-0-8-4	11-0-8-8	12-11-0-8-6	11-0-9-8-4	6
0111	7	BEL 0-9-8-7	ETB 0-9-6	' 8-5	7 7	G 12-7	W 0-6	g 12-0-7	w 11-0-6	11-9-7	12-9-8	12-0-9-8	11-0-9-7	12-0-8-5	11-0-8-9	12-11-0-8-7	11-0-9-8-5	7
1000	8	BS 11-9-6	CAN 11-9-8	(12-8-5	8 8	H 12-8	X 0-7	h 12-0-8	x 11-0-7	0-9-8	9-8	12-0-8-1	11-0-9-8	12-0-8-6	12-11-0-8-1	12-0-9-8-2	11-0-9-8-6	8
1001	9	HT 12-9-5	EM 11-9-8-1	) 11-8-5	9 9	I 12-9	Y 0-8	i 12-0-9	y 11-0-8	0-9-8-1	9-8-1	12-11-9-1	6-8-1	12-0-8-7	12-11-0-1	12-0-9-8-3	11-0-9-8-7	9
1010	10	LF 0-9-5	SUB 9-8-7	* 11-8-4	: 8-2	J 11-1	Z 0-9	j 12-11-1	z 11-0-9	0-9-8-2	9-8-2	12-11-9-2	12-11-0	12-11-8-4	12-11-0-2	12-0-9-8-4	12-11-0-9-8-2	10
1011	11	VT 12-9-8-3	ESC 0-9-7	+ 12-8-6	; 11-8-6	K 11-2	[12-8-2	k 12-11-2	{ 12-0	0-8-4-3	9-8-3	12-11-9-3	12-11-0-9-1	12-11-8-2	12-11-0-3	12-0-9-8-5	12-11-0-9-8-3	11
1100	12	FF 12-9-8-4	FS 11-9-8-4	, 8-6-3	< 12-8-4	L 11-3	\ 0-8-2	l 12-11-3	| 12-11	0-9-8-4	12-9-4	12-11-9-4	12-11-0-9-3	12-11-8-3	12-11-0-4	12-0-9-8-6	12-11-0-9-8-4	12
1101	13	CR 12-9-8-5	GS 11-9-8-5	- 11	= 8-6	M 11-4	] 11-8-2	m 12-11-4	} 11-0	12-9-8-1	11-9-4	12-11-9-5	12-11-0-9-3	12-11-8-4	12-11-0-5	12-0-9-8-7	12-11-0-9-8-5	13
1110	14	SO 12-9-8-6	RS 11-9-8-6	. 12-8-3	>' 0-8-6	N 11-5	^ 11-8-7	n 12-11-5	~ 11-0-1	12-9-8-2	9-8-6	12-11-9-6	12-11-0-9-4	12-11-8-5	12-11-0-6	12-11-0-8-2	12-11-0-9-8-6	14
1111	15	SI 12-9-8-7	US 11-9-8-7	/ 0-1	? 0-8-7	O 11-6	_ 0-8-5	o 12-11-6	DEL 12-9-7	11-9-8-3	11-0-9-1	12-11-9-7	12-11-0-9-5	12-11-8-6	12-11-0-7	12-11-9-8-3	EO 12-11-0-9-8-7	15

CARD READER/CARD READER-PUNCH DEVICE-SPECIFIC IORB FIELDS

Table 6-6 defines the device-specific fields in the IORB not previously defined. Refer to "Driver Functions and Function Codes" earlier in this section.

Table 6-6. Card Reader/Card Reader-Punch IORB Fields

IORB Word	Field	Definition	Use
I_CT2	Function code	0 = Wait online	See "Wait Online Function" earlier in this section.
		1 = Write	Driver "writes" card for "range" number of bytes.
		2 = Read	Driver "reads" card for "range" number of bytes.
		3 = Write file mark	Driver "writes" end-of-file card.
		A = Connect B = Disconnect	See "Connect Function" and "Disconnect Function" earlier in this section.
I_RNG	Range	0≤range≤32K-1	If range is greater than card size, residual range reflects the difference.
I_DVS	Device-specific	0 12 13 14 15 0 0 0 mode mode: 0=ASCII 2=verbatim	Defines character set of data being read.
I_RSR	Residual range	0≤ initial range	Detects device malfunction.

CARD READER/CARD READER-PUNCH HARDWARE STATUS CODE MAPPING

The card reader/card reader-punch controller returns to the driver various codes, which are made visible to the application by way of the IORB as shown in Tables 6-7 and 6-8.

Table 6-7. Card Reader IORB Hardware/Software Status Code Mapping

Hardware Status	IORB Word I_ST	Meaning If Bit Set
0	-	Device ready
1	-	Attention
2	2	Data service rate error
3	3	Mark sense mode
4	4	40-column card mode
5	5	51-column card mode
6	6	External clock track
7	7	Read check error
8	8	ASCII code error
-	-	
-	-	
-	-	
12	-	Corrected memory error
13	15	Nonexistent resource/fatal error
14	15	Bus parity error/fatal error
15	15	Uncorrectable memory error/fatal error

Table 6-8. Card Reader/Punch Hardware/Software Status Code Mapping

Hardware Status	IORB Word I_ST	Meaning If Bit Set
0	-	Device ready
1	-	Attention
2	2	Data service rate error
3	3	Invalid ASCII code
4	4	Punch echo or read registration
5	5	Light/dark check
6	6	Card jam
7	-	
8	-	
-	-	
-	-	
-	-	
12	-	Corrected memory error
13	15	Nonexistent resource/fatal error
14	15	Bus parity error/fatal error
15	15	Uncorrectable memory error/fatal error

Printer Driver

The printer driver performs all data transfers to all line and serial printers as well as terminal print devices. Format control of printing can be achieved by supplying a control byte as the first entry in a data buffer. The control byte is included in the range count of the IORB for the request. The presence of a control byte is indicated by bit 4 of the IORB's I_DVS word.

PRINT CONTROL BYTE

The format of the control byte is:

Bit:	0	1	2	3	4 7
Field:	Y	PP		V	COUNT

The control byte, if supplied, is interpreted differently by line printer and terminal printer devices. The significance of the control byte for both device types is shown in Table 6-9 under "Action Caused".

Table 6-9. Print Control Byte

Field	Action Caused	
	Line Printer (Space Before Print)	Terminal Printer (Space After Print)
Y	Not used.	0 = Use carriage return and/or line feed in I_DVS. 1 = Ignore carriage return and/or line feed in I_DVS.

Table 6-9 (cont). Print Control Byte

Field	Action Caused	
	Line Printer (Space Before Print)	Terminal Printer (Space After Print)
PP	00 Print; ignore V and count/fields; single space to end-of-form; then skip to head-of-form. 01 Do not print; perform actions defined in V and count fields. 10 Print; perform actions defined in V and count fields. 11 Reserved for system use.	Not used.
V	0 Prespace according to count field. 1 If count = 0, skip to head-of-form. If count is between 1 and 11, and the VFU option is present, skip to the VFU channel defined by the count field. If count is greater than 11, or there is no VFU option, do one prespace.	0 = No prespace. 1 = Prespace three lines; count field must be 0.

Table 6-10 summarizes control byte settings as hexadecimal and ASCII values.

Table 6-10. Print Control Byte Summary

Code		Resulting Action
Hexadecimal	ASCII	
Line/Serial Printers		
00-1F	NUL-US	Single space, then print; skip to head-of-form at end-of-form.
20-2F	Δ - /	Space count lines; do not print.

Table 6-10 (cont). Print Control Byte Summary

Code		
Hexadecimal	ASCII	Resulting Action
	Line/Serial Printers (cont)	
30-3F	0 - ?	Skip to VFU channel number in count, do not print.
40-4F	@ - O	Space count number of lines, print.
50-5F	P - _	Skip to VFU channel number in count, print; 50 = skip to head-of-form.
60-6F	\ - o	Reserved for future use.
70-7F	p - DEL	Reserved for future use.
	Terminal Printers	
00-0F		
20-2F	Δ - /	No prespace, print.
40-4F	@ - O	
10-1F 30-3F 50-5F	$\overline{0}$ - ? P - _	Prespace three lines; print.
60-6F	' - o	Reserved for future use.
70-7F	p - DEL	Reserved for future use.

These conventions permit a control byte (e.g., 41) to be used with a printer driver (whose default I_DVS word is all zeros), and with an ASR/KSR driver (whose default I_DVS word is hexadecimal 30), without extra spacing or overprinting. Both drivers support a terminal format convention that does not require a control byte. This convention treats the first byte of the range as data, with spacing as follows:

ASR/KSR - Print, followed by carriage return and line feed (if specified by I_DVS).

Printer - Space one line or skip to head-of-form if at end-of-form, then print.

Bit 4 (F-bit) in I_DVS controls format selection (see Table 6-23).

PRINTER DEVICE-SPECIFIC IORB FIELDS

Table 6-11 defines the IORB fields whose contents are specific to the printer driver.

PRINTER HARDWARE/SOFTWARE STATUS CODE MAPPING

Table 6-12 indicates the hardware/software status code mapping for printers.

Table 6-11. Printer IORB Fields

IORB Word	Field	Definition	Use
I_CT2	Function code	0 = wait online 1 = write	See "Driver Function and Function Codes". Driver will "write" from I_ADR "range" number of bytes.
I_RNG	Range	0≤range≤32K-1	If range is greater than line size, residual range reflects the difference.
I_DVS	Device-specific	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 0 0 0 F 0 0 0 0 0 0 0 0 0 0 0 F: 0 = Assumes line printer format control (control byte) 1 = Assumes terminal format control (no control byte) All other bits must be zero.	
I_RSR	Residual range	See Note	
I_ST	Software status word	Shown below	Mapped from RCT hardware status.

NOTE

For cases where original range is less than or equal to line length, the value in the residual range has the following meanings:

0 - Completed space/print operation.

other value - Residual spacing value is contained in the I_ST value field.

Table 6-12. Printer Hardware/Software Status Code Mapping

Hardware Status	IORB I_ST	Meaning If Bit Set
0	-	Device ready
1	-	Attention
2	2	Lost data
3	3	End-of-form
4	4	Lines per inch: 0 = 6; 1 = 8
5	5	Protocol error
6	6	Power up
7	7	Eight bit mode
-	-	
-	-	
-	-	
-	-	
12	-	Corrected memory error
13	15	Nonexistent resource/fatal error
14	15	Bus parity error/fatal error
15	15	Uncorrectable memory error/fatal error

Disk Driver

A single disk driver supports the following disk devices: diskette, cartridge disk, Lark disk, cartridge module disk, and mass storage unit.

DISK DRIVER CONVENTIONS FOR DISKETTE

The following driver conventions apply to diskette:

- The disk driver supports both 8 inch and 5 1/4 inch diskette devices. For the 8 inch diskette, both single- and double-sided diskettes may be used. Support of the 5 1/4 inch diskette consists of double-sided and double-density.
- The driver does not explicitly reference the volume ID of the diskette; therefore, the user must ensure that volumes addressed are on the proper drives.
- All sector addresses used in the IORB are relative to track 0/sector 0.
- The driver converts the volume relative sector number, defined in the IORB, into physical track and sector numbers, and to a "side" value for two-sided diskette, which it then sends to the device to define the operation.

- The driver can support more than one diskette device, as long as each device is configured at a different level.

- A diskette sector is 128 bytes long (8 inch) or 256 bytes long (5 1/4 inch). If range is less than sector length, a write command will zero fill the rest of the sector. If range is greater than sector length on either a read or a write, the driver will read/write multiple sectors including switching to the next adjacent track, if necessary.

- There are 16 sectors per track for 5 1/4 diskette; 26 sectors per track for 8 inch diskette.

- There are three models:

  1 track per cylinder:
  77 cylinders

  2 tracks per cylinder:
  77 cylinders
  80 cylinders

- If hardware errors occur, the operation (seek or read/write) will be retried up to eight times (five retries and three retries with recalibrate).

- If the device is not ready, a return status of "device not ready" (5) will be returned.

Diskette IORB Fields

Tables 6-13 and 6-14 define IORB fields specific to diskette. Other IORB fields are described in Table 6-3.

Table 6-13. Diskette IORB Fields

IORB Word	Field	Definition	Use
I_CT2	Function code	0 = wait-online 1 = write data 2 = read data 5 = format write 6 = format read E = read disabled device	Specifies I/O operation.

Table 6-13 (cont). Diskette IORB Fields

IORB Word	Field	Definition	Use
I_DVS	Device-specific	Relative sector number	Driver converts this to physical track number and physical sector number on the track, and to a "side" value for two-sided diskette.
I_ST	Software status	Shown below	Hardware status word from diskette (following I/O).
I_RSR	Residual range	0 ≤ original range	Residual range will always be equal to zero (i.e., transfer completed) unless there is a hardware malfunction, or an invalid track number is supplied during a read or write operation.

NOTE

To ensure compatibility of an application with other devices, clear to zero the IORB words I_RSR and I_ST before making an I/O request.

Table 6-14. Diskette Hardware/Software Status Code Mapping

Hardware Status	IORB I_ST	Meaning if Bit Set
0	-	Device ready
1	-	Attention
2	2	Data service rate error
3	3	Deleted field
4	4	Read error

Table 6-14 (cont). Diskette Hardware/Software Status Code Mapping

Hardware Status	IORB I_ST	Meaning if Bit Set
5	5	Device fault
6	6	Missed data synchronization
7	7	Unsuccessful search
8	-	Two-sided diskette
-	-	
10	10	Seek error
12	-	Corrected memory error
13	15	Nonexistent resource/fatal error
14	15	Bus parity error/fatal error
15	15	Uncorrectable memory error/fatal error

DISK DRIVER CONVENTIONS FOR CARTRIDGE DISK

The following driver conventions apply to cartridge disk:

- Sector size is 256 bytes; there are 24 sectors per track.
- The driver does not explicitly refer to the volume ID of the disk; the user must ensure that the volumes addressed are on the proper drives.
- All sector addresses used in the IORB are relative to cylinder 0, track 0, sector 0.
- The driver converts the volume relative sector number, defined in the IORB, into physical cylinder, track, and sector numbers, which it then sends to the device to define the operation.
- There are two models:

  2 tracks per cylinder:
  204 cylinders
  408 cylinders

- Cartridge disk requires two LRNs, one for the fixed and one for the removable platter.
- Cartridge disk driver logic combines seek and data transfer functions. When errors occur, eight attempts are made to correct an error, four seek/data transfers and four seek/data transfers with recalibrate.

- Offset read capability is provided by specifying the desired displacement in the I_RSR field of the IORB.
- Offset write capabilities are not provided.
- When the driver notes a change in the ready state, it disables the device (by a software switch) and notifies the file manager, which executes the automatic volume recognition procedures.

Cartridge Disk IORB Fields

Tables 6-15 and 6-16 show IORB fields specific to the cartridge disk. Other IORB fields are described in Table 6-3.

Table 6-15. Cartridge Disk IORB Fields

IORB Word	Field	Definition	Use
I_CT2	Function Code	0 = Wait online 1 = Write 2 = Read 5 = Format write 6 = Format read E = Read disabled device	Specifies I/O operation.
I_DVS	Device specific	Relative sector number	Driver converts this to physical cylinder, track, and sector number to locate the data needed.
I_ST	Software status word	See Table 6-16	Hardware status from disk (following I/O).
I_RSR	Residual range	0 ≤ original range	Prior to a read, an offset value may be specified here so that reading can begin at a location other than the physical sector boundary; after I/O operation, the field contains the number of bytes not transferred in the operation.

NOTE

To ensure compatibility of an application with other disk devices, clear to zero the IORB word I_ST before requesting I/O.

Table 6-16. Cartridge Disk Hardware/Software Status Code Mapping

Hardware Status	IORB I_ST	Meaning If Bit Set
0	-	
1	-	
2	2	Over or underrun
3	3	Write protect error
4	4	Read error
5	5	Invalid seek
6	6	Missed data synchronization
7	7	Unsuccessful search
8	8	Missed clock pulse
9	9	Successful recovery
10	10	Seek error
11	-	
12	-	
13	-	
14	-	
15	15	Fatal error

DISK DRIVER CONVENTIONS FOR LARK DISK

The Lark device is a random access, rotating 8-inch disk with both removable and fixed platters.

The following conventions apply to Lark devices:

- Sector size is 256 bytes; there are 64 sectors per track.
- The driver does not explicitly refer to the volume ID of the disk; the user must ensure that the volumes addressed are on the proper drives.
- All sector addresses used in the IORB are relative to cylinder 0, track 0, sector 0.

- There are two models:

  2 tracks per cylinder:
  204 cylinders
  622 cylinders

- The driver converts the volume relative sector number, defined in the IORB, into physical cylinder, track, and sector numbers, which it then sends to the device to define the operation.

- The Lark disk requires two LRNs, one for the fixed and one for the removable platter.

- Offset read capability is provided by specifying the desired displacement in the I_RSR field of the IORB.

- Offset write capabilities are not provided.

Lark Disk IORB Fields

Tables 6-17 and 6-18 show IORB fields specific to the Lark device. Other IORB fields are described in Table 6-3.

Table 6-17. Lark Disk IORB Fields

IORB Word	Field	Definition	Use
I_CT2	Function Code	0 = Wait online 1 = Write 2 = Read 5 = Format write 6 = Format read E = Read disabled device	Specifies I/O operation.
I_DVS	Device specific	Relative sector number	Driver converts this to physical cylinder, track, and sector number to locate the data needed.
I_ST	Software status word	See Table 6-18	Hardware status from disk (following I/O).

Table 6-17 (cont). Lark Disk IORB Fields

IORB Word	Field	Definition	Use
I_RSR	Residual range	0 ≤ original range	Prior to a read, an offset value may be specified here so that reading can begin at a location other than the physical sector boundary; after I/O operation, the field contains the number of bytes not transferred in the operation.
NOTE To ensure compatibility of an application with other disk devices, clear to zero the IORB word I_ST before requesting I/O.			

Table 6-18. Lark Disk Hardware/Software Status Code Mapping

Hardware Status	IORB I_ST	Meaning If Bit Set
0	-	
1	-	
2	2	Over or underrun
3	3	Write protect error
4	4	Read error
5	5	Invalid seek
6	6	Missed data synchronization
7	7	Unsuccessful search
8	8	Missed clock pulse
9	9	Successful recovery
10	10	Seek error
11	-	
12	-	
13	-	
14	-	
15	15	Fatal error

DISK DRIVER CONVENTIONS FOR MASS STORAGE UNIT

The following driver conventions apply to mass storage units:

- Sector size is 256 bytes; there are 64 sectors per track.
- The driver does not explicitly refer to the volume ID of the disk pack, so the user must ensure that the volumes addressed are on the correct drives.
- All sector addresses in the IORB are relative to cylinder 0, track 0, sector 0. There are four models:

  5 tracks per cylinder:
  411 cylinders
  823 cylinders

  19 tracks per cylinder:
  411 cylinders
  823 cylinders

- The driver converts the volume relative sector number, defined in the IORB, into physical cylinder, track, and sector numbers, which it then sends to the device to define the disk address.
- The volume relative sector numbers exceed the maximum number that may be stored in one I_DVS word. Place high order bits in I_ST; low order bits in I_DVS.
- The mass storage unit requires only one LRN.
- The driver combines seek and data transfer functions. When errors occur, eight attempts are made to correct the error: five seek/data transfers, and three seek/data transfers with recalibrate.
- Offset read capability is provided by specifying the required displacement in the I_RSR field of the IORB.
- Offset write capability is not provided.
- When the driver notes a change in the ready state, it disables the device (by a software switch) and notifies the file manager to execute the automatic volume recognition procedures.

Mass Storage Unit IORB Fields

Tables 6-19 and 6-20 show IORB fields specific to the mass storage unit. Other IORB fields are described by Table 6-3.

Table 6-19. Mass Storage Unit IORB Fields

IORB Word	Field	Definition	Use
I_CT2	Function Code	0 = Wait online 1 = Write 2 = Read 5 = Format write 6 = Format read E = Read disabled device	Specifies I/O operation.
I_DVS	Device-specific	Relative sector number	Driver converts this to the physical cylinder, track, and sector number to locate the data needed.
I_RSR	Residual range	0 ≤ original range	Prior to a read, an offset value may be specified here so that reading can begin at a location other than the physical sector boundary; after I/O operation the field contains the number of bytes not transferred in the operation. After an I/O operation, the field contains the number of bytes not transferred.
I_ST	Software status word	See Table 6-20	Prior to an order, this field contains the high-order bits of the relative sector number. After the operation, it contains the hardware status from device.

Table 6-20. Mass Storage Unit Status Code Mapping

Hardware Status	IORBV I_ST	Meaning If Bit Set
0	-	
1	-	
2	2	Over/underrun
3	3	Device fault
4	4	Read error
5	5	Invalid seek
6	6	Missed data synchronization
7	7	Unsuccessful search
8	8	Missed clock pulse
9	9	Successful recovery
10	10	Reserved
11	-	
12	-	
13	-	
14	-	
15	15	Fatal error

DISK DRIVER CONVENTIONS FOR CARTRIDGE MODULE DISK

The following driver conventions apply to the cartridge module disk:

- Sector size is 256 bytes; there are 64 sectors per track.
- The driver does not explicitly refer to the volume ID of the disk; the user must ensure that the volumes addressed are on the correct drives.
- All sector addresses in the IORB are relative to cylinder 0, track 0, sector 0. The models are:

  1 track per cylinder:

  411 cylinders (removable, 8-megabyte)
  411 cylinders (fixed, 8-megabyte)
  823 cylinders (removable, 16-megabyte)
  823 cylinders (fixed, 16-megabyte)

  3 tracks per cylinder:

  823 cylinders (removable, 16-megabyte)
  823 cylinders (fixed, 48-megabyte)

5 tracks per cylinder:

823 cylinders (removable, 16-megabyte)
823 cylinders (fixed, 80-megabyte)

The driver converts the volume relative sector number, defined in the IORB, into physical cylinder, track, and sector numbers, which it then sends to the device to define the disk address.

- The volume relative sector numbers exceed the maximum number that may be stored in I_DVS word; place high order sector bits in I_ST, low order sector bits in I_DVS.
- The fixed and removable portions of the cartridge module disk each require a separate LRN.
- The driver combines seek and data transfer functions. When errors occur, eight retries are made (five seek/data transfers, three seek/data transfers with recalibrate).
- Offset reading (not writing) is provided by specifying the required displacement in the I_RSR field of the IORB.
- When the driver detects a change in the ready state, it disables the device, both fixed and removable (with a software switch), and notifies file management to execute the system's automatic volume recognition procedures.

Cartridge Module Disk IORB Fields

Tables 6-21 and 6-22 show the IORB fields specific to the cartridge module disk. Other IORB fields are described by Table 6-3.

Table 6-21. Cartridge Module Disk IORB Fields

IORB Word	Field	Definition	Use
I_CT2	Function Code	0 = Wait online 1 = Write 2 = Read 5 = Format write 6 = Format read E = Read disabled device	Specifies I/O operation.

Table 6-21 (cont). Cartridge Module Disk IORB Fields

IORB Word	Field	Definition	Use
I_DVS	Device-specific	Relative sector number	Driver converts this to the physical cylinder, track, and sector number to locate the data needed.
I_RSR	Residual range	0 ≤ original range	Prior to a read, an offset value may be specified here so that reading can begin at a location other than the physical sector boundary; after I/O operation the field contains the number of bytes not transferred in the operation.
I_ST	Software status word	See Table 6-22	Prior to an order, this field contains the high-order relative sector bits. After the I/O operation, the field contains the hardware status, from the device.

Table 6-22. Cartridge Module Disk Status Code Mapping

Hardware Status	IORB I_ST	Meaning If Bit Set
0	-	
1	-	
2	2	Over/underrun
3	3	Device fault
4	4	Read error
5	5	Invalid seek
6	6	Missed data synchronization
7	7	Unsuccessful search
8	8	Missed clock pulse
9	9	Successful recovery
10	10	Reserved
11	-	
12	-	
13	-	
14	-	
15	15	Fatal error

ASR/KSR and CONSOLE Drivers

The CONSOLE driver includes all the functionality described below for the ASR/KSR driver. In addition, the CONSOLE driver must be used to perform forms processing from an MDC-connected VIP7200 or VIP7205 terminal.

The keyboard/printer functions of an ASR are supported; the paper tape reader/punch functions are not. Thus, the K-bit within I_DVS word (Table 6-23) must be zero.

To examine the first character of a message sent in single character mode (from a local KSR terminal) before the rest of the message is transmitted, proceed as follows:

1. Issue a single character asynchronous read with no echo to the terminal.

2. When the read is complete, examine the character; then if the rest of the message is wanted, write the character to the terminal (with no carriage return or line feed).

3. Issue a read for the rest of the message (with echo).

Note that the operator terminal (keyboard/printer), when used, must be configured at LRN=0. For information about dialogue with the operator's terminal, see the System User's Guide.

Character codes, function codes, and device control available for the keyboard/printer are described in the following paragraphs.

KEYBOARD INPUT

- Keyboard input is accepted until end-of-range, or carriage return, whichever occurs first. The carriage return character is not included as part of the input data.
- Keyboard control (line feed, carriage return, etc.) is definable in the IORB.
- Editing characters can control input:
  - @ Deletes the previous character entered.
  - CTL X Deletes all the previous characters entered on the same input line.
  - \ Character immediately following is treated as input.

NOTES

1. When CTL X is struck, the characters *DEL* are displayed on a separate line. Further input may begin after completion of the *DEL* output.
2. The back slash character (\) causes the character immediately following to be treated as data and not an editing character; the backslash itself is not placed in memory.

PRINTER OUTPUT

- Printer output is accepted until end-of-range.
- Timeout period for keyboard/printer operation is 5 minutes.

ASR/KSR IORB FIELDS

Tables 6-23 and 6-24 show IORB fields specific to ASR/KSR devices. Other IORB fields are described by Table 6-3.

Table 6-23. ASR/KSR IORB Fields

IORB Word	Field	Definition Keyboard/ Printer	Use
I_CT2	Function code	1 = write 2 = read 9 = break notification A = Connect B = Disconnect	Used by driver to complete the description of the requested I/O function.
I_DVS	Device-Specific	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 B 0 0 S F T 0 Q D K E L C M A H Bit — Value B (When function code is 9; i.e., break control request) 0 = Request break notification. 1 = Abort previous break notification requests. S 0 = Function code 2 implies normal terminal read. 1 = Function code 2 implies single-character mode read. F 0 = Assumes line printer format control. 1 = Assumes terminal format control. T 0 = Use control characters in control functions. 1 = Treat all characters as data; bypass control character checks on input.	

Table 6-23 (cont). ASR/KSR IORB Fields

IORB Word	Field	Definition Keyboard/ Printer Use
I_DVS (cont)		Bit Value Q 0 = Stop output immediately on detecting "attention" when the detected character has No Stop bit status (e.g., a "break" key). 1 = Post "attention" and allow completion of output transfer. D 0 = Read attention character with input (if present). 1 = Discard attention character on input. K 0 = Transfer to keyboard/printer. (Must be 0.) E 0 = Do not echo keyboard input. 1 = Echo keyboard input. L 0 = No line feed at end of transfer. 1 = Issue line feed after transfer. C 0 = Issue carriage return after transfer. 1 = No carriage return after transfer. M 0 = Transfer mode is 7-bit, with parity. 1 = Transfer mode is 8-bit direct transcription mode.

Table 6-23 (cont). ASR/KSR IORB Fields

IORB Word	Field	Definition Keyboard/ Printer	Use
I_DVS (cont)		Bit	Value
		A	In single-character mode: 0 = Do not abort previously buffered single-character mode characters in queue. 1 = Abort previously buffered single-mode characters in queue.
		A	On disconnect: 0 = Abort I/O requests on disconnect. 1 = Do not abort I/O requests on disconnect.
		H	0 = Disconnect with phone hang up. 1 = Disconnect without phone hang up.
		NOTE The MDC-connected ASR/KSR driver does not check bit H.	
I_ST	Software status word	Shown below	Mapped by driver from the hardware status to tell requesting task the hardware status of the I/O operation.

Table 6-24. ASR/KSR Hardware/Software Status Code Mapping

Hardware Status	IORB I_ST	Meaning If Bit Set
0	-	Device ready
1	-	Attention
2	2	Data service rate error
3	3	Parity error (even)
-	-	
5	5	No stop bit
-	6	Long record
-	7	Checksum error
8	8	Control character number 2 termination
9	9	Control character number 3 termination
-	-	
-	-	
12	-	Corrected memory error
13	15	Nonexistent resource/fatal error
14	15	Bus parity error/fatal error
15	15	Uncorrectable memory error/fatal error

Magnetic Tape Driver

The magnetic tape driver manages all standard data transfer requests to and from 9-track phase encoded (PE), and 9-track nonreturn to zero inverted (NRZI) tape drives on one or more magnetic tape controllers. The tape drive characteristics supported by this tape driver are shown in Table 6-25.

Table 6-25. Characteristics of Supported Tape Drives

Tape Drive Type	Speed (ips)			Density (bpi)					Parity	
	45	75	125	6250	1600	800	556	200	Odd	Even
9-track NRZI	X	X	-	-	-	X	X	-	X	-
9-track PE	X	X	-	-	X	X	-	-	X	-
9-track GCR GCR mode	-	X	X	X	-	-	-	-	X	-

Table 6-25 (cont). Characteristics of Supported Tape Drives

	Speed (ips)			Density (bpi)					Parity	
Tape Drive Type	45	75	125	6250	1600	800	556	200	Odd	Even
9-track GCR PE mode	-	X	X	-	X	-	-	-	X	-

The driver provides the following callable functions:

- Wait online
- Write
- Read (forward)
- Position block (forward and backward)
- Position forward or backward by tape mark, rewind to beginning of tape (BOT), rewind to BOT and unload.

The driver operates in the following modes:

- Odd parity
- Minimum data block, MDB (American National Standard specifies 18 or more characters per block in write, 12 or more in read)
- MDB-inhibited (if fewer than the specified number of characters must be read or written, this mode is required).

If MDB mode is specified for a write and the range is less than 18 characters, a parameter error is reported. If MDB mode is specified for a read and the range is less than 12 characters, you receive the first portion (requested range) of the first valid block and an unequal length check. If a "short record" is detected, a corrected media error is reported in status word, I_ST. If a record of less than 18 characters is written or less than 12 characters is read, the inhibit block size check bit (bit 12 of the device specific word, I_DVS) must be set.

Beginning of tape (BOT), end of tape (EOT), and end of file (EOF) conditions are reported for appropriate user action. If an error occurs in a case when the operation can be retried, the driver backspaces and reissues the order up to 32 times before reporting a hardware error. If an error occurs and no retry is possible, the driver rewinds and forward spaces to the problem block and reissues the order once before reporting a hardware error. The driver does not check the tape volume identifier.

The EOT return status is not returned for read operations; only the EOT status word bit is set. It is assumed that appropriate application software conventions will prevent reads that would force the tape off the end of the reel.

The resident magnetic tape driver is interrupt driven and must execute with a resident Executive and with the central processor in the privileged state. It can support, on an adapter, one data transfer simultaneously with one or more rewind/rewind-unload orders.

MAGNETIC TAPE IORB FIELDS

Tables 6-26 and 6-27 show IORB fields specific to magnetic tape devices. Other IORB fields are described by Table 6-3.

Table 6-26. Magnetic Tape IORB Fields

Word	Field	Definition
I_CT2	Function code	0 = Wait online 1 = Write 2 = Read 3 = Write filemark 4 = Position by block (see range) 6 = Position file (see range)
I_DVS	Device-specific	0 12 13-15 0 0 0 0 0 0 0 0 0 0 0 0 0 I mode I: 0 = Normal American National Standard block sizes 1 = Inhibit sensing for American National Standard block size mode: Must be zero.

Table 6-26 (cont). Magnetic Tape IORB Fields

Word	Field	Definition
I_RNG	Range	Write: 1 through 7FFF Read: 0=Backspace one block; 1 through 7FFF is valid for data transfer Position by block: Negative is backspace; 0 is invalid Positive is forward space Position by file: -2 = Rewind and unload -1 = Rewind to BOT 0 = Backspace to previous tapemark 1 = Forward space to tapemark
I_RSR	Residual range	Nonzero when physical block exceeds range.

Table 6-27. Magnetic Tape Hardware/Software Status Code Mapping

RCT R_STTS	IORB I_ST	Meaning If Bit Set
0	-	Device ready
1	-	Attention
-	1	Rewinding
2	2	Error - Operation can be retried
3	-	Must be zero
-	3	Write protected
4	4	Corrected media error
5	5	Tape mark
6	6	BOT
7	7	EOT
8	8	Unequal record length
9	9	Error - Operation cannot be retried
10	10	Must be zero
11	11	Operation check
12	-	Corrected memory error
-	12	High density
13	15	Nonexistent resource/fatal error
14	15	Bus parity error/fatal error
15	15	Memory error - correction impossible/fatal error

Section 7
LINE PROTOCOL HANDLERS

This section provides an overview of line protocol handlers. Subsequent sections describe specific line protocol handlers in detail.

LINE PROTOCOL HANDLERS

A communications protocol is a set of conventions or rules for the transmission of data. Communications protocols are used in the transfer of information between a local CPU and remote terminal or host CPU.

A line protocol handler (LPH) is the implementation of a particular communications protocol. Accordingly, each LPH supports a specific class of communications device, such as synchronous VIP terminals, or a communications protocol, such as the 2780/3780 binary synchronous communications protocol.

The following LPHs can be configured at system building:

ATD

The asynchronous terminal driver (ATD) supports asynchronous terminals, serial printers, and certain asynchronous data streams. The ATD LPH has five operational modes: Teletype compatible (TTY), field, block, receive-only printer (ROP), and stream.

STD

The synchronous terminal driver (STD) LPH supports specific synchronous terminal devices. These devices are the polled visual information projection (VIP) terminals and associated ROPs.

PVE

This synchronous LPH supports communications between computers. It emulates the polled VIP protocol for use in communications with remote Honeywell hosts that support polled VIP terminals.

2780/3780 BSC

This synchronous LPH supports communications between computers. It supports a station (device or computer) that utilizes the 2780 or 3780 binary synchronous communication (BSC) protocol in communications with a remote host.

TTY

This asynchronous LPH supports specific asynchronous terminal devices. These devices are classified as teleprinter-compatible, and include certain automatic send/receive (ASR), keyboard send/receive (KSR), and VIP terminals.

The user may write a line protocol handler if it conforms to the same internal interface requirements used by the Honeywell-supplied line protocol handlers.

LINE PROTOCOL HANDLER FUNCTIONS

Line protocol handlers transfer data between a communications device and the application that uses it. These handlers consist of two parts -- one resident in main memory and the other (called the channel control program (CCP)) resident in the MLCP.

The main memory-resident portion of the LPH is concerned with the processing of transmitted/received data at the block, message, or field level. The MLCP-resident component is concerned with the transmission/reception of the individual data characters that make up the block, message, or field level data aggregate.

Main Memory-Resident LPH

The portion of the line protocol handlers resident in main memory performs the following:

- When the system is bootstrapped:
  - Validates communication device types by reading the device's identification number.
  - Initializes the communication device and sets it to the priority level at which it is to operate.
- Validates the application's input/output request block (IORB) fields.
- Converts user-supplied functions into device-specific MLCP orders.
- Sets a timer and a monitor for data set status changes.
- Initiates the MLCP I/O operation.
- Detects and processes MLCP I/O interrupts.
- Reads return status from the communication device to ascertain result of an I/O operation.
- Processes error recovery, when possible.
- Processes unsolicited timeouts and data set status changes.
- Forms composite status in the IORB, including residual range, from all of the processed MLCP orders.
- Posts back the application's IORB with the appropriate hardware and software status information.

MLCP-Resident LPH (CCP)

A channel control program (CCP) is the MLCP-resident portion of an LPH.

Through the appropriate hardware device-pac attached to the MLCP, the channel control program controls transmission of data over communication lines. It serves to:

- Store or fetch individual characters in or from the buffer supplied with the IORB
- Perform translation, substitution, and deletion operations on individual characters
- Insert/delete protocol or device-specific header or trailer information.

MLCP COMMUNICATIONS HANDLER

The MLCP communications handler receives processor orders from the main memory-resident portion of the line protocol handler and activates the appropriate channel control program (see above and Figure 7-1) to process the orders. The handler also:

- Processes a line protocol handler's requests for control functions or for data transfer operations
- Services interrupts from the MLCP and passes them to the appropriate line protocol handler.

COMMUNICATIONS SUBSYSTEM OPERATION EXAMPLE

The following example and Figure 7-1 indicate the interaction of the communications subsystem's components in the processing of a connect, write, and disconnect request. The operations described apply to the physical I/O interface, without reference to a specific device or line protocol.

This example refers to the communications supervisor. The communications supervisor resides in main memory and provides the interface to communications applications programs at the physical I/O level. It queues application programs' requests for services, activates the appropriate line protocol handler, interacts with an application through system software when an I/O order is complete, and provides a set of common line protocol handler services (e.g., establishing/disestablishing data set communications, monitoring for time-outs and data set status changes).

Example:

1. The communications supervisor receives the application's connect request through the physical I/O interface, and passes it to the DIAL channel control program (CCP) within the multiline communications processor (MLCP).

2. The DIAL CCP establishes a physical communication connection to the device.

3. The main memory-resident portion of the appropriate line protocol handler (LPH) processes the logical connection.

4. The communications supervisor passes the application's subsequent write request to the main memory-resident LPH, which translates the request into one or more MLCP communications handler requests.

5. Each MLCP communications handler request results in one or more orders to the MLCP. (These orders not only describe the data to be transferred, but also cause the invocation and execution of the appropriate CCP.)

6. The appropriate CCP processes each of the write orders, which transmits the data to the device. During this time, the main memory-resident LPH terminates itself.

7. When the MLCP senses completion of the data transfer, the CCP issues an interrupt, which is processed first by the communications supervisor and then by the MLCP communications handler.

8. The MLCP communications handler reactivates the main memory-resident portion of the LPH at the interrupt level, to minimally process the interrupt.

9. When processing is complete, control passes to the MLCP communications handler, which causes processing at the interrupt level to be suspended.

10. If additional processing is necessary, the main memory-resident portion of the LPH can schedule itself to perform post-interrupt processing on a non-interrupt level.

11. The application's disconnect request is processed in the same manner as the connect request, but in the opposite order.

    a. The main memory-resident portion of the LPH performs the necessary logical disconnect processing.

    b. The physical connection is appropriately disconnected by the DIAL CCP.

The logic of the write operation in this example would apply to a read operation.

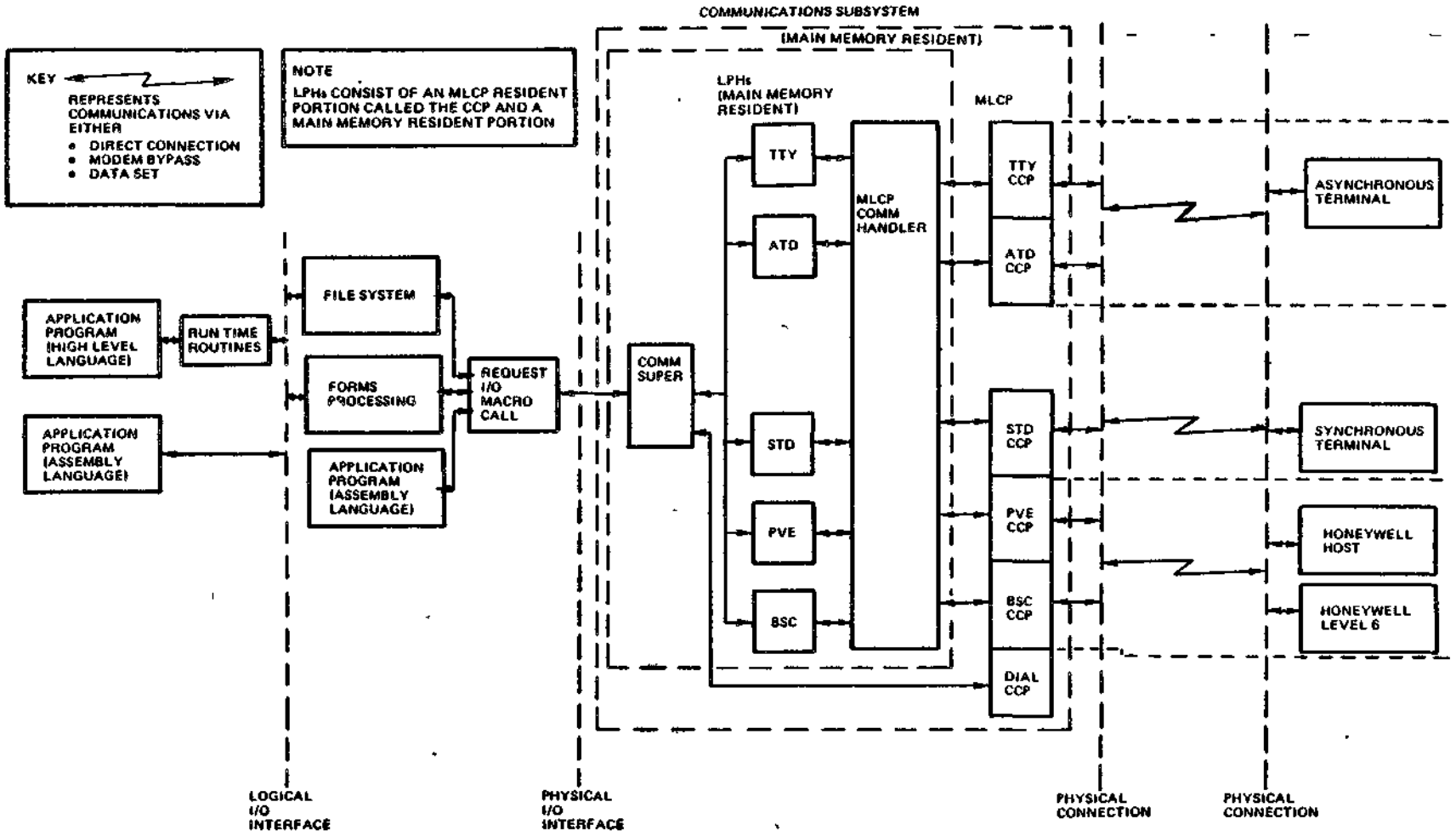


Figure 7-1. Communications Overview

MODEM SUPPORT

For asynchronous devices, the communications subsystem provides the following modem support:

- Bell System Data Sets: Types 103A, 113F, 202, 212A
- Honeywell modem bypass
- Any modem type defined by the user at system building
- Honeywell-supplied direct-connect cables.

For medium speed synchronous communications, the communications subsystem provides the following modem support:

- Bell System Data Sets: Types 201A, 201B, 201C, 203, or 208A
- Honeywell modem bypass
- Honeywell-supplied direct connect cables
- Any modem type defined by the user at system building time.

For high-speed synchronous communications, the communications subsystem provides the following modem support:

- Bell System Data Sets: Types 301B or 303.

AUTO CALL UNIT

When congfigured into the system, the Auto Call Facility uses an Auto Call Unit (ACU) to initiate a line connection with a remote auto answer data set. The facility operates in the following manner:

1. The user associates the Auto Call Unit with a particular communications channel at system building time by using the ACU CLM directive.

2. The user enables the Auto Call Facility by setting bit 2 of the I_DVS word to one on a connect request. The facility is supported by all LPHs.

3. When the connect request is processed, the system attempts to dial a line, using a list of telephone numbers supplied at system building, the first entry of which is null. The first number to be dialed can then be specified with a Set Dial ($SDL) macro call or with the Set Autodial Telephone Number (SDL) command. If the first number on the list is not specified (by the macro call or command), the system skips to the next number on the list.

4. The facility dials each number on the list three times at 40-second intervals until the list is exhausted or a connection made, whichever occurs first.

5. The facility checks that a connection to the modem has been made.

6. When the connection has been made, control is passed to the LPH, which processes the logical portion of the connect request.

The Auto Call Unit supports Data Auxiliary Set Automatic Calling Units 801A and 801C. The ACU adapter and the adapter for its associated data line must be on the same controller.

Two data set options are required to use the Auto Call Unit:

- The option that terminates the call, through the data set, after the DSS (data set status change) goes on

- The option that stops the ACR timer when the DSS goes on.

COMMUNICATIONS SUBSYSTEM ERROR AND CORRECTION PROCEDURES

The communications subsystem detects errors that may occur over communications lines by means of parity checking, block checking, and timeout checking.

Parity Error Check

The system sends a parity (check) bit with each transmitted character. The parity bit, plus the number of character bits set to 1, will always be an odd or even-numbered total for every character, according to whether transmission is odd parity (total is an odd number) or even parity (total is an even number).

The ATD and TTY line protocol handlers support parity error checking.

Block Error Check

The communications subsystem uses two kinds of block error checking: the longitudinal redundancy check (LRC) and the cyclic redundancy check (CRC). The computed check characters are known as block check characters (BCC).

LONGITUDINAL REDUNDANCY CHECK (LRC)

The LRC is a simple check that is applied to the entire message. The system appends an LRC character, which is an exclusive OR of all the characters in the message, to the end of every message.

The STD and PVE line protocol handlers use the LRC method.

CYCLIC REDUNDANCY CHECK (CRC)

The CRC method is also block-oriented. The system computes the CRC block check character(s), using special algorithms applied to the data to be checked. The system then appends the BCC to the message.

Only the BSC line protocol handler uses the CRC method of checking errors.

BSC BLOCK CHECK CHARACTER (BCC)

In ASCII transmission, the 8-bit BCC is the result of an exclusive OR operation on all bits transmitted, beginning with the first character following the STX and ending with the ITB , ETB, or ETX control character. It is based on the polynomial:

$$X^8 + 1 .$$

In EBCDIC transmission the BCC is 16 bits, and is calculated by the system with the checking polynomial:

$$1 + X^2 + X^{15} + X^{16} .$$

Timeout Check

After sending a message, the LPH waits for an acknowledgment from the receiving device. When there is no acknowledgment after a specific interval, the LPH retransmits the message.

When there is no acknowledgment after a specified number of transmissions, the LPH takes whatever action is specified by the protocol.

Section 8
ATD LINE PROTOCOL HANDLER

INTRODUCTION

The Asynchronous Terminal Driver (ATD) line protocol handler supports certain asynchronous terminals, serial printers, and certain types of asynchronous data streams.

The ATD LPH operates in five modes:

- TTY mode, which supports line-at-a-time transfer of data to or from any teletype compatible (TTY) terminal.
- Field mode, which supports field and forms processing on VIP7200, VIP7800, and VIP7300 class terminals.
- Block mode, which supports transfer of blocks of data to or from any VIP7800 class terminal.
- ROP mode, which supports output to certain receive-only printers (ROPs).
- Stream mode, which supports transfer of data on any asynchronous line that utilizes an X-ON/X-OFF flow-control protocol. Typically, such a line is associated with paper tape readers and punches.

The ATD LPH can be accessed at the Physical I/O or File System level. At the Physical I/O level, the LPH is accessed through the Request I/O ($RQIO) macro call and an associated input/output request block (IORB). This interface can be used with any mode of the LPH and provides for complete control of the selected mode.

The LPH is accessed indirectly through the File System. For example, to read input from a terminal, an application issues a Read Record macro call, supplying parameters for the call in an associated file information block (FIB). The File System translates the macro call and FIB parameters into a $RQIO macro call and associated read IORB. The File System interface is most useful in providing a sequential file interface to terminals (operating in TTY and block mode), serial printers (operating in ROP mode), and paper tape devices (operating in stream mode). The File System interface does not support field mode.

The remainder of this section provides:

- A summary of ATD operational modes
- A description of common functions
- A detailed description of each mode.

ATD MODES

A particular mode is selected by means of a connect IORB and remains in effect until a disconnect IORB is received. The following subsections indicate the uses of each mode.

TTY Mode

TTY mode is the default ATD operating mode. The user need not specify this mode in the connect IORB device specific word (DSW). This mode is used primarily by the File System, which treats a terminal (configured by means of the DEVICE directive) as a sequential file. In this mode, a terminal can be used as the input and output file of a task group (i.e., user-in, user-out, command-in, error-out).

TTY mode provides for line-at-a-time input and output. Character-cancel, line-delete, input-terminator, and escape key functionality is provided to aid the operator in data entry operations at the terminal. Support is also provided for a break key. (The terminal keys that represent these functionalities can be redefined by the terminal operator through the Set Terminal File Characteristic (STTY) command.)

TTY mode supports a great variety of asynchronous terminals including VIP7100, 7200, 7207, 7801, 7803, 7808, 7301, 7303, 7307; TWU1001, 1003, 1005; TN0300, 1200, and other teletype (KSR, ASR) terminals.

Field Mode

Field mode allows forms-oriented processing to be performed (on certain terminals) by applications such as Display Formatting and Control (DFC), menu subsystem, and Data Entry Facility (DEF). A form consists of a series of fields. A field is a series of contiguous locations on the terminal screen into which only selected types of data can be entered. For example, a terminal operator can enter only "0" through "9" into a numeric field. The validation of data entered into a field is accomplished by ATD under application control.

Field mode allows the operator to modify entered fields easily. The break key is configurable by means of the STTY command. Break or supervisory messages are displayed in a communications region (line) on the terminal screen.

Field mode processing is limited to the following terminals: VIP7200, 7207, 7801, 7808, 7301, 7303, and 7307.

Block Mode

Block mode is supported by the VIP7800 series of terminals. In block mode, the operator can locally edit terminal input without ATD involvement. Depression of the transmit key causes the LPH to receive data from the terminal in blocks of fully-edited input. Block mode can be used at either the Physical I/O or File System level.

Terminal input is locally edited by means of cursor control, character insertion/deletion, and line insertion/deletion keys. Termination of input is accomplished by depression of the transmit key. The break key is configurable by means of the STTY command. When the terminal is operating in no-roll mode, supervisory messages can be displayed in a communications region (line) on the terminal screen.

Block mode processing is limited to the following terminals: VIP7801, 7803, and 7808.

ROP Mode

ROP mode supports selected serial and letter-quality receive-only printers (ROPs). This mode provides full control-byte processing; it also detects and analyzes, in some cases, printer off-line conditions. ROP mode is supported at the Physical I/O or File System level.

ROP mode is limited to the following serial printers: PRU1004, 7007, 7070, and 7075.

Stream Mode

Stream mode allows an application to use a paper-tape reader or punch that utilizes an X-ON (DC-1)/X-OFF (DC-3) flow control protocol. The mode can also be used by two co-operating applications for the high-speed transmission (up to 9600 characters per second) of data over an asynchronous communication line.

Control byte processing enables File System applications to directly control the operation of stream mode.

Stream mode requires at least a half-duplex or, in some cases, a full-duplex communications line.

Stream mode is supported at the Physical I/O or File System level.

I/O FUNCTIONS SUPPORTED BY ATD

The ATD line protocol handler supports five logical functions. Each is listed below with its associated function code (fc).

- Connect (fc = A)
- Disconnect (fc = B)
- Read (fc = 2)
- Write (fc = 1)
- Define form, field mode only (fc = 5)
- Break (fc = 9).

These functions are requested through the input/output request block (IORB). An application places in the right byte of IORB word I_CT2 the code of the desired function. A connect request establishes the mode in which subsequent functions (e.g., read, write) are performed.

IORB PROCESSING

The ATD LPH is activated by an application-generated $RQIO macro call. Associated with this macro call is an input/output request block (IORB) that specifies the operation to be initiated. The IORB contains a function code, a buffer address, and range (in most cases), and parameters that specialize execution of the requested operation.

Figure 8-1 shows a representative IORB, as required for field mode processing.

WORD	LABEL	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1_LNK	RESERVED FOR SYSTEM USE AS POINTER															
$AF	1_CT1	RETURN STATUS								T	W	U	S	0	R	0	1
1+$AF	1_CT2	LHN								0	B	0	E	FUNCTION			
2+$AF	1_ADR	BUFFER ADDRESS SAF 1-WORD POINTER LAF 2-WORD POINTER															
2+2*$AF	1_RNG	RANGE - NUMBER OF BYTES TO BE TRANSFERRED															
3+2*$AF	1_DVS	DEVICE-SPECIFIC WORD															
4+2*$AF	1_RSR	RESIDUAL RANGE - NUMBER OF BYTES NOT TRANSFERRED															
5+2*$AF	1_ST	DEVICE STATUS WORD															
6+2*$AF	1_EXT	TOTAL IORB EXTENSION LENGTH (IN WORDS)								PHYSICAL EXTENSION LENGTH (IN WORDS)							
7+2*$AF	1_DV2	DEVICE-SPECIFIC WORD 2															
8+2*$AF	1_FCS	TOTAL KEYSTROKES															
9+2*$AF	1_HDR	READ OFFSET															
10+2*$AF	1_ST2	FIELD MODIFICATION INDICATOR															
11+2*$AF	1_QDP	DEVICE_ID; RELATIVE RESIDUAL RANGE															
12+2*$AF	1_TAB	EDIT OFFSET (INPUT); TERMINATION CHARACTER (INPUT)															
13+2*$AF	1_CON	ABSOLUTE ADDRESS INDICATOR; PRE-ORDER READ AND WRITE CODE; TERMINATION CHARACTERS; VFN VALUE															
14+2*$AF	1_LOG	START OF FIELD ATTRIBUTE TABLE															

Figure 8-1. ATD IORB

IORB Size

The required size of an IORB depends upon the mode selected by the application. Field mode requires that an extended-length IORB be used for all orders (including connect). If a standard length read or write IORB is received when the terminal is connected in field mode, that IORB is treated as a supervisory message.

The other ATD modes require standard-length IORBs. Extended IORBs can optionally be used when connecting a terminal in block mode to ascertain the terminal's type (which is returned in the extended portion of the IORB).

IORB Device-Specific Word

The device-specific word I_DVS is used in conjunction with each of the I/O functions. This word serves to modify the activity of a particular function. For example, the setting of bit 15 in I_DVS determines whether the communication line is disconnected on completion of a disconnect function.

Processing Order of IORBs

An application can issue one I/O order against a terminal (or line) and wait for its completion, or issue several IORBs. Outstanding read and write orders and non-abortive disconnects are queued sequentially. In TTY, field, and block mode, write orders are processed before read orders if the read order is not in progress. Define form orders, read and write orders with the option to purge outstanding I/O requests, and abortive disconnects are executed immediately after being received by the LPH.

Purging Queued IORBs

In the following cases, the LPH purges queued IORBs and posts the incomplete orders back to the requesting application:

1. The application issues a disconnect order with an abort request (purge IORB indicator in I_DVS word of IORB is set to 0). All read and write orders that are active or queued at the time of the disconnect order are purged and posted to the issuing task with a "device unavailable" (010B) return status.

2. A line disconnect (data set status change) occurs. All active or queued read and write orders are purged and posted with a "device unavailable" return status. Both the line and station are disconnected.

3. The application issues a purge-all order in field mode. All active or queued read and write orders are purged and posted to the issuing task with a "device unavailable" return status. Both the line and station remain connected.

4. A break signal is detected (BREAK key pressed) and the user has previously issued a read-break IORB (i.e., function code 9 in I_CT2, and bit 0 in I_DVS set to 0). See "Break Processing by ATD LPH" below.

5. The application issues a block write order with the purge option. Active or queued write orders are purged or posted with "device unavailable" return status. Both the line and station remain connected.

6. The application issues a block read order with the purge option. Active or queued read orders are purged or posted with "device unavailable" return status. Both the line and station remain connected.

IORB Error Processing

All ATD modes report errors in the same manner. A 2-byte error code is placed in register R1. The left byte indicates the component detecting the error; the right byte indicates the error itself. The right byte is also placed in IORB field I_CT1. Table 8-1 lists the return codes as they appear in the left byte of I_CT1.

Table 8-1. ATD Return Codes

Status Byte	Meaning
0	No error; operation complete
1	Request block is already busy
2	Invalid LFN
3	Illegal wait
4	Invalid argument(s): • Improper set-up of IORB • Improper buffer size • Improper set-up of data in certain buffers
5	Device not ready. Reported when the following devices are in an off-line state: TWU1001, 1003, 1005; PRU7070, 7075; and serial printer attached to VIP7800 terminal
6	Timeout on order other than connect

Table 8-1 (cont). ATD Return Codes

Status Byte	Meaning
7	Hardware error: • Parity error (block mode) • Framing error • Data service error (receive overrun) • Communications control block service error • Fatal MLCP error
8	Device disabled • Connect or disconnect pending • Device logically disabled by system
A	Controller unavailable
B	Device unavailable • Read/write IORBs purged by purge option • Read/write IORBs purged by disconnect • Read/write IORBS purged by disconnect with queue abort • Attempt made to connect to a 7800 class terminal that is in local mode
C	Inconsistent or illogical request • Connect order issued against a device that is currently connected • Disconnect order issued against a device that is currently disconnected • Read/write IORB issued; line not connected • Connect order issued to VIP7800 attached printer when terminal has already been connected in field mode • Connect order issued to VIP7800 terminal when attached printer has already been connected • Field mode read issued before define form request • Read request outstanding when new define form request issued • Block missed on block mode read
F	End of file detected (stream mode)
10	Timeout on connect

The status word (I_ST) of the IORB contains additional information that qualifies the major status code returned in I_CTl. The significance of certain bits of the status word is the same for all ATD modes. Table 8-2 shows the meaning of these bits.

Table 8-2. Status Word of IORB (I_ST)

I_ST bit	Meaning When Bit Set to 1
0	Mode specific
1	Mode specific
2	Data service rate error (receive overrun)
3	Mode specific
4	Communication control block service error
5	IORB purged because of break signal
6	Mode specific
7	Mode specific
8	Framing error
9	Parity error
A	Nonzero residual range (read only)
B	Phone hang-up on disconnect
C	Mode specific
D	Mode specific
E	Mode specific
F	Fatal error • Unrecoverable memory error • Bus parity error • Non-existent resource error

For the significance of mode specific bit settings, refer to the descriptions of the individual ATD modes found later in this section.

Return of Device ID

Table 8-3 shows the values returned in the right byte of IORB field I_QDP when a field or block mode, extended length, connect IORB completes and is posted back to the application.

Table 8-3. Device IDs Returned in IORB

Value in I_QDP	Marketing Identifer
45	VIP7100
46	VIP7200
47	VIP7207
49	VIP7801
4A	VIP7808
4B	VIP7803
4C	TTY
4D	TN 0300
4E	TN 1200
51	TWU1003
52	TWU1005
57	TWU1001
5F	PRU7075
65	VIP7307
66	VIP7303

SUPERVISORY MESSAGE PROCESSING

When a terminal is processing forms, the supervisory message line provides a communication region (typically the bottom line of the terminal) through which the operator can interact with the system independently of the forms processing application. ATD provides support for supervisory messages on the following VIP terminals when they are connected in either field or block mode: VIP7200, 7207, 7801, 7803, 7808, 7301, 7303, and 7307.

Supervisory message processing is specified by means of a non-extended read or write IORB with bit 9 in I_DVS set to 1. The use of this bit is optional in ATD field mode, because supervisory message orders are already distinguished from normal field mode orders by being non-extended.

The location of the supervisory message line depends on the ATD mode, and the type and operational mode of the terminal. When, for example, a terminal is connected in field mode, it operates in no-roll mode. If the terminal is a VIP7200, then the supervisory message line is (typically) the 24th line. If, however, the terminal is a VIP7801, then the supervisory message line is always the 25th line of the terminal.

The following diagram shows supervisory message line location for supported VIP terminal classes and ATD LPH modes.

	VIP Terminal Class			
	7100	7200	7300	7800
TTY mode	1	1	1	1
Field mode	N/A	2	3	3
Block mode	N/A	N/A	N/A	4

where:

1 = When supervisory messages are written to the terminal, output is at current cursor position. There is no way to acknowledge the write; all reads are treated as normal device reads.

2 = Read/write activity is directed to the designated supervisory message line, which is normally line 24.

3 = Read/write activity is directed to the line 25.

4 = Read/write operation is predicated on the roll bit (bit 9) of the connect I_DVS. If the terminal is in roll mode (bit 9 = 0), writes begin at the current cursor position and reads are treated as normal device reads. If the terminal is not in roll mode (bit 9 = 1), reads and writes are directed to the 25th line.

All writes to the supervisory message line are truncated to 80 characters.

If supervisory message writes are specified, bit 8 of the read/write I_DVS becomes significant. If bit 8 = 0, supervisory messages must be acknowledged before the write is posted back to the application. If bit 8 = 1, supervisory messages need not be acknowledged by the operator.

In TTY mode, supervisory writes (which are treated as standard data writes) are not acknowledged. In other modes, the operator acknowledges a supervisory message by pressing one of the following keys:

Field mode: function key 10, CLEAR key, or transmit key.
Block mode: function key 10.

CONTROL BYTE PROCESSING

Control byte processing is a TTY, block, and ROP mode option that is specified by a bit setting in the I_DVS word of the write order. When selected, this option indicates that the first byte of the output buffer is to be used as a control byte. This byte must be included in the range (I_RNG) value of the write IORB.

An application uses the control byte to cause either a head-of-form sequence or from one to fifteen line feeds to precede the display of data. If an application specifies both head-of-form sequence and line feeds, only head-of-form sequence occurs.

If the terminal supports form feed, as do most serial printers, the head-of-form sequence is a form feed. Otherwise, as with most video terminals, the head-of-form sequence is a carriage return followed by three line feeds.

The first bit of the control byte indicates whether any post order carriage return and/or line feed(s) specified in I_DVS are to be carried out or ignored (i.e., overridden by the control byte).

Bits in the control byte have the following meaning:

Bit 0:

0 = Perform post-order LF/CR specified in I_DVS
1 = Ignore post-order LF/CR specified in I_DVS

Bits 1-2:

00 = Ignore bits 3-7
XX = Process bits 3-7

Bit 3:

0 = Do not generate head-of-form sequence
1 = Generate head-of-form sequence

Bits 4-7:

Number of lines feeds to be generated; a value from 1 through fifteen; value ignored if bit 3 = 1

The head-of-form sequence, specified by bit 3, is a form feed for the following devices: TWU1001, 1003, 1005; PRU 7070, 7075, 1004, 7007. These are stand-alone devices not attached to a VAF7821 buffered printer adapter. For other devices, head-of-form consists of a carriage return and three line feeds.

BUFFERED PRINTER ADAPTER (BPA) SUPPORT

ATD supports the buffered printer adpater (BPA). The BPA (also called the VAF7821) allows the attachment of a serial printer (PRU1003, 1005, or 7075) to a 7801, 7803, or 7808 VIP terminal. An application can use the serial printer when the attached terminal is connected and operating in either TTY or block mode. Use of the printer with a terminal connected in field mode is not allowed.

Configuring the BPA

The BPA can be accessed at the physical I/O or File System level. It must be configured with the ASP directive. If accessed through the File System, the ASP directive must be paired with a DEVICE directive specifying a ROP device_unit.

Connecting the BPA

Before issuing write orders to the BPA, the application must first establish a connection to it. Upon completion, the application's connect order will be posted with one of the following status codes in the left byte of I_CTl:

0 - Connect complete

C - Attempt to connect BPA when terminal is connected in field mode

5 - Attached serial printer is powered off or in an off-line state

Writing to the BPA

To use the BPA at the physical I/O level, the application issues I/O orders to the work station with a single LRN that refers to the terminal display/keyboard and the BPA. A sub-LRN specified in I_ST differentiates between orders directed to the terminal display/keyboard and to the BPA. A sub-LRN of 0 refers to the display /keyboard; a sub-LRN of 1 refers to the BPA.

When the attached printer is servicing a write order, the terminal keyboard is locked. If the write order specifies a control byte, only a head-of-form sequence (a carriage return followed by three line feeds) is supported.

A write order is posted with one of the following status codes:

0 - Write complete

5 - Attached serial printer is powered off or is in an off-line state.

BREAK PROCESSING BY ATD LPH

In TTY, field, and block mode, break processing is initiated when the terminal's BREAK (BRK) key is pressed. Results differ, depending on whether the task issued a read-break I/O order request for that terminal.

Break Processing with Read Break Request

A task issues a read break request when the IORB specifies a function code value of 9 in I_CT2 and a value of 1 in bit 0 of I_DVS. I_ADR of the IORB must have a null address.

The communications supervisor queues read break requests on a last-in, first-out basis.

When the terminal's break key is pressed, and a read break request has been issued, the terminal is now in "break mode" for subsequent I/O requests. Break processing proceeds as follows:

1. When a write order is active, and:

   a. Bit 7 in I_DVS of the write IORB is 1, the order completes normally; break processing then begins with step 2 below

   b. Bit 7 in I_DVS of the write IORB is 0, or when a read order is active, either order is terminated and posted to the issuing task with IORB settings shown in step 2.

2. All other queued read and/or write IORBs are posted back to their respective tasks with:

   a. I_RSR containing the range value specified in I_RNG

   b. Bits 5 and 10 of I_ST set to 1

   c. Left byte (status) in I_CT1 has value of 0.

3. The last (last-in, first-out) read break request is posted to the issuing task with:

   a. Bit 5 of I_ST set to 1

   b. Left byte (status) of I_CT1 has value of 0.

4. Read and write orders issued by the "broken task" (i.e., task in break mode) are posted back (without execution) with IORB values described in step 2 above.

5. Read and write orders from tasks not in break mode (i.e., that did not issue receive-break requests) are accepted and executed.

Break mode remains in effect until a task issues another read break request or a cancel break request (i.e., until provision has been made for processing the next break signal). A task issuing another read break request to a device which is in break mode is indicating that it wishes to be the task notified of the next break. A task issuing a cancel break request to a device which is in break mode is indicating that it does not wish to be the task notified of the next break; the task to be notified of the next break is the one that issued the most recent read break order.

A cancel break request is specified with an IORB having a function code of 9 in I_CT2 and bit 0 of I_DVS set to 1. A cancel break request causes one or all queued read break IORBs to be posted back to their issuing tasks. If bit 1 of I_DVS is 0, the request specifies the cancellation of only the most recently issued read break request. If bit 1 of I_DVS is 1, the request specifies the cancellation of all active and queued read break requests. The cancel break IORB and purged read break IORB(s) are posted back to their issuing tasks with:

Bit 5 in I_ST1 set to 0
Left byte (status) in I_CT1 set to 0.

Break Processing with No Read Break Request

When a break signal is recieved and no read break request has been issued, only the current active order is affected. The break signal is processed as follows:

1. If there is no active order, the break signal is ignored.

2. When a read order is active, the order is terminated and posted to the issuing task with:

   a. I_RSR containing the range value specified in I_RNG
   b. Bits 5 and 10 of I_ST set to 1
   c. Left byte of I_CT1 set to 0.

3. When a write order is active and bit 7 in I_DVS is 1, the break signal is ignored and the write order completes normally.

4. When a write order is active and bit 7 in I_DVS is 0, the order is posted to the issuing task with:

   a. I_RSR containing the range value specified in I_RNG
   b. Bits 5 and 10 of I_ST set to 1
   c. Left byte of I_CT1 set to 0.

TTY MODE

The TTY mode of ATD provides for line-at-a-time transfer of data to or from teletype-compatible asynchronous terminals.

TTY mode supports five functions:

- Connect
- Disconnect
- Read
- Write
- Break.

These functions are requested through standard-length IORBs. An application can optionally use an extended IORB for a connect operation.

A connect order establishes the mode in which the connected terminal operates. Because TTY is the default mode of the ATD LPH, an application need not explicity specify the mode in the device-specific word (I_DVS) of the connect IORB.

Connect Function (TTY Mode)

The following paragraphs describe the options that an application can specify with a connect order.

AUTO CALL

The Auto Call option, which is supported by all system-supplied LPHs, is described in Section 7. This option enables an application to establish a connection with an 801-A or 801-C ACU data set.

BELL

The default IORB setting for this option allows the output of bells to a terminal. If the option is not specified, the output of bells to a terminal is suppressed, even under error conditons.

CHARACTER/BUFFERED

When the terminal being connected is a VIP7801, 7803, or 7808, specification of character mode (which is the default) causes the terminal to be physically configured in character mode with the echoplex and roll options set.

When the buffered option is selected, a VIP7800 class terminal is configured in text mode with the no-echoplex and no-roll options set. This means that data entered at the terminal is not transmitted (to the LPH) until the transmit key is depressed. Prior to pressing the transmit key, the operator can edit information displayed on the terminal by means of the cursor control and erase keys. When ATD receives and processes the transmitted data, the LPH acts on any line cancel or character delete sequence encountered in the data stream. That is, the LPH does not accept as data the @, \, or CTL-X characters. This point bears emphasis; the operator of a buffered terminal who uses the cursor-back key to erase a character might well forget that pressing the @ key has the same effect. If the operator mistakenly enters the @ character as data, the LPH deletes the next character when data is ultimately transmitted from the terminal. Care must be exercised when entering teletype control sequences from a buffered terminal.

Connect IORB (TTY Mode)

This subsection summarizes the bit settings that govern the connect options already described.

BIT SETTINGS OF I_DVS

Table 8-4 shows bits of the connect I_DVS word that are applicable to TTY mode.

Table 8-4. I_DVS Word in Connect IORB (TTY Mode)

Bit Number	Bit Value	Meaning for Connect Function
2	0	Do not use auto dial
	1	Use auto dial
3	0	Allow output of bells to the terminal
	1	Supress output of bells to the terminal
13	0	Character mode
	1	Buffered mode

BIT SETTING IN WORD I_ST

This field is signficant when a serial printer is attached to the terminal by means of a VIP7800 buffered printer adapter (VAF7821). On connect orders, the field specifies whether the terminal or attached printer is being addressed. The permitted values are:

0 = Terminal
1 = Attached serial printer.

Disconnect Function (TTY Mode)

An application uses the disconnect IORB to terminate TTY mode processing.

The following paragraphs describe the options that an application can specify with a disconnect order.

ABORT QUEUED ORDERS

If the abort option is specified, outstanding IORBs (active and queued) are terminated with a "device unavailable" status (010B). The disconnect order is immediately serviced. If the abort order is not specified, all outstanding IORBs are allowed to complete before the disconnect order is serviced.

HANG UP

If the hang-up option is selected, the terminal is physically disconnected when the disconnect order is serviced. If the hang-up option is not specified, the communications connection remains active after servicing of the disconnect order (i.e., the terminal is logically disconnected, but remains physically connected).

Disconnect IORB (TTY Mode)

This subsection summarizes the IORB bit settings that govern the disconnect options just described.

BIT SETTINGS OF I_DVS

Table 8-5 shows bits of the disconnect IORB that are applicable to the TTY mode of ATD.

Table 8-5. I_DVS Word in Disconnect IORB (TTY Mode)

Bit Number	Bit Value	Meaning for Disconnect Function
14	0	Abort outstanding requests
	1	Wait until outstanding requests complete before disconnecting terminal
15	0	Hang up the phone
	1	Do not hang up the phone

BIT SETTING IN WORD I_ST

This field is signficiant when a serial printer is attached to the terminal by means of a VIP7800 buffered printer adapter (VAF7821). On disconnect orders, the field specifies whether the terminal or printer is being addressed. The permitted values are:

0 = Terminal
1 = Attached serial printer.

Read Function (TTY Mode)

The following TTY mode read functions support the entry of data by the terminal operator. They are activated by pressing terminal keys. In some cases, an application can designate the key that activates a particular function by means of the Set Terminal File Characteristics (STTY) command. These functions are not controllable through the IORB. The read IORB is used to pass data to the application once it has been entered and edited by the operator.

OPERATOR FUNCTIONS

TTY mode functions that support data entry operations are the following:

Function	Action
Character delete	Delete a previously entered character
Line cancel	Cancel the current line of input
Hide	Accept the next character as data (i.e., do not interpret it as a control character)
Terminate read	Signal completion of the current read order
Break	Generate break signal to application controlling the terminal

Operator Function Keys

The LPH performs one of the functions just listed when the operator keys the appropriate code sequence. Typically, the depression of a single terminal key will generate the proper code sequence. For example, on a VIP7301 terminal, depression of the cursor-left key causes the generation of the code sequence 1B44, which causes the LPH to delete the prior character.

The code sequence that initiates a function is determined by the device-type parameter of the ATD directive. That code sequence can later be altered by the STTY command.

Table 8-6 shows the initial (default) codes associated with device-types that can be specified with the ATD directive.

Table 8-6. Default Values of Special Characters by Device Type

Device Type	Character Delete		Line Cancel		Line Break		Read Terminator	
	Hex	Key-Cap	Hex	Key-Cap	Hex	Key-Cap	Hex	Key-Cap
VIP7200	1B44	<-	1B4B	ERASE	00	BREAK	0D	RETURN
VIP7207	1B44	<-	1B60	CLEAR	00	BREAK	0D	RETURN
VIP7301	1B44	<-	1B4B	ERASE	00	BREAK	0D	RETURN
VIP7303	1B44	<-	1B4B	ERASE	00	BREAK	0D	RETURN
VIP7307	1B44	<-	1B60	CLEAR	00	BREAK	0D	ENTER
VIP7801	1B44	<-	1B4B	ERASE	00	BREAK	0D	RETURN
VIP7803	1B44	<-	1B4B	ERASE	00	BREAK	0D	RETURN
VIP7808	1B44	<-	1B4B	ERASE	00	BREAK	0D	RETURN
VIP7100	40	@	18	CTL-X	00	BREAK	0D	RETURN
TWU1001	40	@	18	CTL-X	00	BREAK	0D	RETURN
TWU1003	40	@	18	CTL-X	00	BREAK	0D	RETURN
TWU1005	40	@	18	CTL-X	00	BREAK	0D	RETURN
TN 0300	40	@	18	CTL-X	00	BREAK	0D	RETURN
TN 1200	40	@	18	CTL-X	00	BREAK	0D	RETURN
TTY	40	@	18	CTL-X	00	BREAK	0D	RETURN

Character Delete and Line Cancel

As the preceding table indicates, the operator can delete characters and cancel lines in two different ways, depending upon the device type. On some devices, referred to in this context as hard copy terminals, deleting a character requires depressing the @ key. On other devices, the cursor back (<-) key is used; these devices are called video terminals.

On hard copy terminals, cancelling a line is accomplished by depressing and holding the CTL key and pressing X. On video terminals, the operator uses the ERASE or CLEAR key.

On hard copy and video terminals, editing is performed by different actions, and different information is displayed at the terminal during the editing operation. However, the modification of buffer contents and the information returned in the IORB is the same. The following paragraphs explain in detail the procedure and process of editing on each type of terminal.

Character Deletion on Hard Copy Terminals. Character deletion is performed on the current line (i.e., before the carriage return key is pressed). Pressing the @ key deletes the character immediately preceding the @ character, and, if echo was requested, displays the @ character. Each succeeding @ entry deletes another character, from right to left, up to the beginning of the line.

The I_RSR value in the issuing program's IORB indirectly reflects the number of characters accepted at the time the order was terminated. For example, if the operator enters AXC@@B followed by a carriage return, the I_RSR value shows that only two characters (A and B) were entered. Note that pressing the @ key does not actually delete a character, but moves back by one character position a pointer in the read buffer. In the example just given, X is overwritten by B, but C (though rejected by the operator and not reflected in the I_RSR value) is present in the buffer, following B.

Line Cancellation on Hard Copy Terminals. To cancel the current line (before carriage return is entered), the operator depresses and holds the CTL (control) key and presses X. This action deletes the current line, displays the *DEL* message on the next line. The LPH reissues the read order, using the original buffer and range. Line cancellation does not clear the buffer of characters entered into the buffer before the line cancellation action.

Character Deletion on Video Terminals. Pressing the cursor-left (<-) key erases from the screen the character last entered, and removes it from the associated read buffer. When the completed read IORB is posted to the issuing application, I_RSR indirectly reflects the number of characters accepted when the order was terminated. For example, if the operator enters ABC<-, I_RSR shows only that two characters (A and B) were entered. Again, as with character deletion on hard copy terminals, extraneous information may appear in the rest of the buffer.

Line Cancellation on Video Terminals. The key used is either ERASE or CLEAR, depending on the device type (see Table 8-6). The effect is to erase all characters on the current line and to reposition the cursor to the beginning of the erased line. The LPH reissues the read order, using the original buffer and range. Line cancellation does not clear the buffer of characters entered into the buffer before the line cancellation action.

Read Termination

The operator can terminate a read order in one of three ways.

1. Press the transmit key.

2. Press the user-selectable read-termination key. The carriage return key is the default termination key on both hard-copy and video terminals. The operator can designate another key by means of the STTY command. The terminating character (generated by carriage return or a user-designated key) is not stored in the buffer; the LPH optionally echoes a carriage return and/or line feed to the terminal.

3. Generate a two- or three-character escape sequence. Any terminal function key or cursor control key generates a two- or three-character escape sequence. This sequence can be used to terminate a read operation, provided that it has not previously been designated for line cancel, character delete, or break operations. ATD stores the terminating sequence in the read buffer and optionally echoes a carriage return and/or line feed, as appropriate. The read IORB is posted back to the application.

Break

The break key provides an interruption or attention signal to the system software. After detecting a break, the LPH may terminate write orders and read orders. For a detailed description of break functionality, see "Break Processing with Read Break Request" earlier in this section.

The break key can be changed by means of the STTY command.

Hide Function

The hide function allows the operator to enter as data a character (such as @, carriage return, and cursor-left) that the LPH would otherwise interpret as a control character. The hide function key is a backslash (\). The operator keys a backslash immediately before the character to be entered as data. The LPH interprets the backslash as an escape character (i.e., does not place the backslash in the buffer) and echoes the backslash, if echo was requested. The LPH then stores the next character in the buffer without interpretation, echoing it if echo was requested. If the hidden character (immediately following the backslash) is not printable, it is still stored in the buffer, but a period (.) is echoed to the terminal.

The backslash key is used for the hide function on hard-copy and video terminals. The hide function key cannot be changed by the STTY command.

READ ORDER FUNCTIONALITY

The following options, unlike those just described, are not under direct control of the operator. Instead, they are specified by the application in an IORB.

Echo

If the echo option is selected, any keyed input is echoed, or "reflected" back to the terminal. If echo is not selected, keyed input will not be echoed and the cursor will not move as the operator enters data at the terminal.

Line Feed

If this post order option is selected by the application, a line feed is sent to the terminal upon completion of a read order. A line feed is not echoed if the read IORB specified the no echo or the no line feed option.

Carriage Return

If this post order option is selected by the application, a carriage return is sent to the terminal upon completion of a read order. A carriage return is not echoed if the read IORB specifies the no echo or the no carriage return option.

READ IORB (TTY MODE)

An application specifies the options just described by setting bits in the IORB word I_DVS. Table 8-7 gives the individual significance of these bits.

Table 8-7. ATD Word I_DVS in TTY Mode Read IORB

Bit Number	Bit Value	Meaning for Field Read Function
10	0	Do not echo input or move the cursor
	1	Echo input; move cursor
11	0	Do not send post-order line feed
	1	Send post-order line feed
12	0	Send post-order carriage return
	1	Do not send post-order carriage return

Write Function (TTY Mode)

The following options are specified by an application in the write IORB.

OFF LINE

If the off-line option is specified, the LPH detects and reports a device-not-ready condition (0105) when a TWU1003 or 1005 is disconnected or non-operational. If this option is not specified, ATD does not detect or report off-line conditions.

CONTROL BYTE PROCESSING

If specified, the control byte option indicates that the first byte in the output buffer is to be used for pre-order control. A control byte must be included in the range (I_RNG) of data to be transmitted. For a detailed description of this option, including control byte format, see "Control Byte Processing" earlier in this section.

QUIT ON BREAK

If this option is specified, a break signal can interrupt the execution of the write order. Otherwise, a break signal cannot be used to prematurely terminate an active write order.

CARRIAGE RETURN

If the carriage return option is specified, a carriage return is sent to the terminal after the completion of the write order.

LINE FEED

If this option is specified, a line feed is sent to the terminal after the completion of the write order.

Write IORB (TTY Mode)

This subsection summarizes the bit settings that govern TTY mode write options.

BIT SETTINGS IN WORD I_DVS

Table 8-8 gives the significance of the bits in the IORB word I_DVS that are applicable to TTY mode ATD.

Table 8-8. ATD Word I_DVS in TTY Mode Write IORB

Bit Number	Bit Value	Meaning for TTY Write Function
2	0	Do not check for TWU1003, 1005 off-line conditions
	1	Check for TWU1003, 1005 off-line conditions
4	0	Include control byte
	1	Do not include control byte
7	0	Stop output on detection of a break
	1	Do not stop output on detection of a break
11	0	Do not send post-order line feed
	1	Send post-order line feed
12	0	Send post-order carriage return
	1	Do not send post-order carriage return

BIT SETTING IN WORD I_ST

This field is signficant when a serial printer is attached to the terminal by means of a VIP7800 buffered printer adapter (VAF7821). On write orders, the field specifies whether the terminal or printer is being addressed. The permitted values are:

0 = Terminal
1 = Attached serial printer

Device Configuration (TTY Mode)

Hardware switches on a device connected in TTY mode should be set in the following positions. (The device may not support all of the switches mentioned below).

TTY (character) mode:

CHARACTER/BUFFER switch in CHARACTER position

DUPLEX HALF/FULL SWITCH in FULL positon

LOCAL COPY/ECHO switch set as required by user (normally set to echo)

Speed configured between 110 and 9600 bits per second

ROLL/NO ROLL switch set to ROLL

Error Processing

When a parity error is detected in keystroke input, an audible alarm sounds and the typed character is ignored. When the read order is posted, the return status in I_ST indicates detection of parity error(s) (bit 9 = 1).

If a framing error or receive overrun conditon is detected, the read order terminates and a hardware error (0107) is returned; I_ST indicates the specific reason for abnormal termination.

TTY Mode Timeout Processing

Timeouts may occur during the processing of read orders. A timeout occurs when the operator does not terminate the input operation within 5 minutes after entering the first character. There is no timeout if the operator does not enter any characters.

Write orders do not incur timeouts.

FIELD MODE

The field mode of ATD allows an application to process a set of fields, commonly called a form. In this mode, each field that an operator keys into the form is validated by the ATD LPH and is passed to the application, one field at a time. This mode should not be used if the terminal itself is performing (local) field validation and forms processing.

The following subsections define the concepts of forms, fields, subfields, and field validation.

Forms, Fields, and Subfields

A field is a series of contiguous locations into which meaningful data can be entered. A subfield is a portion of a field (less than or equal to the field size) that accepts data only in accordance with the definition of the subfield

There are no limits on the number of fields that a form may contain. Each field may contain one to nine subfields. A field may not be longer than 80 characters and may not extend over one line (row) of the terminal display area.

An example of the relationship between field and subfield is an 8-character alphanumberic employee ID consisting of a 5-character employee number and a 3-character department designator. The first subfield would be defined as 5 digit characters and the second subfield as 3 alphabetic characters.

INPUT VALIDATION

The input to a subfield is validated by reference to a field attribute descriptor. A subfield descriptor must specify one of the following validation/edit attributes:

- Digit (0-9)
- Numeric (0-9, decimal point, minus sign, plus sign, comma)
- Alphabetic (A-Z, a-z, period, space, comma, hyphen, apostrophe)
- Alphanumeric (all numeric and alphabetic)
- No validation (95-character code set equivalent to the last 6 columns of the ASCII table, excepting DEL, note that the hyphen and minus sign are the same ASCII character, as are the period and decimal point).

When an invalid character is entered into a subfield requiring validation, an audible alarm is sounded, the cursor remains in its current position, and the character is not accepted or echoed. The LPH continues to process the current order without notifying the application of the input error. When the field is completed and accepted by the LPH, further validation may be performed by the application.

For reasons of security, an application may specify (in I_DVS) no echo for a field. When an invalid character is entered into such a field, no audible alarm is sounded.

AUTO-INSERT CHARACTERS

An auto insert character is a predetermined character in a predetermined location within a field. It is defined as a subfield by the field attribute descriptor.

Consider, as an example, the standard Social Security account number:

123-45-6789

This field occupies 11 positions. It can be defined as an 11-character numeric field, in which case the operator must key in the hyphen. It can also be defined as follows:

A digit subfield of 3 positions
An auto-insert character
A digit subfield of 2 positions
An auto-insert character
A digit subfield of 4 positions

In this case, the operator may not key in anything but digit characters. The hyphens are inserted automatically by the LPH.

Restrictions

Contiguous auto-insert subfields are not allowed; at least one other type of subfield must be defined between auto-insert subfields within a field. An auto-insert must not be the first or last subfield of a field.

SEPARATE SIGN FIELD

The separate sign subfield allows the operator to enter a minus or plus sign as the first character of a field. If a character other than a minus or plus sign is entered, a plus is assumed and placed in the buffer associated with the field read order. The keyed character is then stored in the buffer. If echo is requested, the assumed plus sign, followed by the keyed character, is displayed on the screen.

If the operator moves the cursor to the left into a separate sign subfield, a new value (+ or -) may be entered. However, if the operator enters another character or moves the cursor right into the separate sign subfield, the default sign (+) is stored in the buffer and displayed on the screen (assuming specification of echo).

Restrictions

The separate sign subfield must be the first subfield of the field. It may only be used in conjunction with a decimal-point and digit subfields.

MUST RELEASE FIELD

Must release fields are the same as normal fields with one exception: the field is not considered complete at end-of range; the operator must key in a terminator character. Take, for example, a form containing two fields. One field is a zip code, defined as digit, length five; the other field is the customer name, defined as alphabetic, length twenty. In a data entry environment, the zip code would probabably not be defined as a must release field; after the operator keys in the five digits, the cursor automatically moves to the next field. The customer name field, however, would probably be defined as a must release field, forcing the operator to key in a terminator character regardless of the length of the customer name. (Valid termination characters are defined later in this section under "Termination of Field".)

If the operator fails to enter an appropriate termination character after filling a field (i.e., after entering twenty alphabetic characters, in the preceding example), an audible alarm sounds until a valid terminator character is entered.

DECIMAL POINT AND DECIMAL POINT PROCESSING

If the decimal point subfield is used, the separate sign must also be specified. The separate sign subfield must be the first subfield of the field. The decimal point subfield must occur somewhere later in the field description and is used by the LPH as an aligment position. The decimal point subfield must not occupy the last position of the field and only one such subfield can be used within a field.

If the operator keys in a plus or minus sign as the first character of a field, the sign is stored in the read buffer and transmitted to the terminal (assuming that echo is specified in the IORB). If the operator keys in any other character except the decimal point as the first character, that character is stored as the second character of the field (following successful validation). It too is echoed to the terminal if echo is specified. If the operator keys in the decimal point character, or if the cursor occupies the position in the field designated for the decimal point, the decimal point character is stored in the buffer at the next available position. It is also transmitted to the screen, assuming specification of echo. The next character entered is treated as part of the next digit subfield following the decimal point subfield, and is validated according to the attributes of that subfield. The operator is not allowed to move the cursor left into an designated decimal point position. An audible alarm is sounded if this is attempted.

Restrictions

This attribute must be used in conjunction with the separate sign and digit subfields. Also, there can be only one occurrence of this subfield and it cannot occupy the last position of the field.

FIELD DESCRIPTOR AND DEFINE FORM

Before a read order in field mode can be processed, the application must either issue a define form request or incorporate a field descriptor in the IORB itself. Bit 2 of I_DV2 indicates whether the IORB is carrying the integrated field descriptor along with the read request. If the bit is on, the field descriptor starts at offset I_LOG in the IORB. Alternatively, with bit 2 of I_DV2 set off, the application must issue a define form order that points to a set or table of field attribute descriptors that define the form.

Integrating a field descriptor in the IORB is the preferred approach, because an application can more efficiently alter an integrated descriptor than one that is part of a external table. After altering the attributes defined by a integrated descriptor, the application issues a single read order; after altering the attributes defined by a descriptor in a table, the application

must issue a new define form order and a field read order. Two I/O orders are required rather than one.

USING THE INTEGRATED FIELD ATTRIBUTE DESCRIPTOR

When using the integrated field attribute descriptor, the application must specify in words the total extension length of the IORB. The integrated descriptor begins at offset I_LOG, which is the first word of the logical part of the IORB. The value for the total size of the IORB extension must include both the size of the physical IORB extension (seven words) and the size of the integrated field attribute descriptor.

USING DEFINE FORM

The following conventions apply to the use of the define form order and the associated table of field attribute descriptors.

1. The IORB that requests a define form order is physically extended.

2. The define form order must be issued before any read order that refers to the field attribute table pointed to by the define form order.

3. After a define order is issued referencing a field attribute table, subsequent define form orders may not be issued while read orders that reference the initial field attribute table are outstanding. The define form order remains active and the associated attribute table is used for all subsequent field reads until another define form or a disconnect order is issued, or a line disconnect is detected.

4. The table address is passed in I_BAD of the define form IORB. The range (I_RNG) must specify the length of the table in bytes. The logical portion of the IORB (I_FCN through I_CON) must be zero.

5. The attribute table must begin on a word boundary; consequently, the buffer bit (bit 8) of I_CT2 must be zero.

6. Once the field attribute descriptor table and its address have been established, any subsequent field read order must specify in I_TAB the word offset to the desired field attribute descriptor. Accordingly, all field attribute descriptors must start on a word boundary.

7. The application may organize the attribute table in any manner that is convenient (as long as the descriptors start on word boundaries). The descriptors may be interspersed with other information, if conservation of memory is not a prime consideration.

8. Conservation of memory can be acheived by the following measures:

   a. If the attributes of two or more fields are exactly alike, only one descriptor is needed. All read orders referring to the identical fields would reference the same descriptor.

   b. In some cases, it might be advantageous to apportion the descriptors describing a form into a set of attribute tables rather than into a single table. Only one table of the set would be in memory at a time; when another attribute table was needed, the application would issue another define form order.

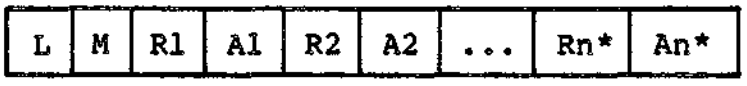
FORMAT OF THE FIELD ATTRIBUTE DESCRIPTOR

Field attribute descriptors have a single format, whether integrated into a field read IORB or belonging to an field attribute descriptor table.

A field may contain one to nine subfields. The field attribute descriptor consists of the following:

- A one-byte entry defining the length of the field descriptor
- A one-byte entry defining the must-release attribute
- A two-byte entry defining the type and range of the subfield (there can be up to nine such subfield definitions)
- A two-byte field descriptor terminator.

The format of these field descriptor components is shown in the following diagram.



where:

L = Length of field descriptor (in bytes), not including this byte; a hexadecimal value in the range 5 to 15.

M = Must release field. Bit 4, when set to 1, signifies that the entire field is designated a must release field. The other bits are reserved for future use and must be zero.

Entries L and M constitute a 2-byte descriptor header.

R = Range of a subfield, in decimal, or zero

A = Attribute of the subfield; a hexadecimal value

Rn* = The value of the last two R and A entries must be zero,
An* to indicate the end of the descriptor. These two entries constitute the terminator.

If the value of a range byte (R) is greater than zero and less than or equal to eighty, the value of the attribute byte (A) has the following significance:

Value	Meaning
00	No validation
10	Digit (0-9)
30	Numeric (0-9, decimal point, minus sign, plus sign, comma)
40	Alphabetic (A-Z, a-z, period, space, comma, hyphen, apostrophe)
70	Alphanumeric (all numeric and alphabetic)

If the value of a range byte (R) is zero, the value of the attribute byte (A) has the following significance:

Value	Meaning
00	End of field
20-7E	Auto-insert character
80	Separate sign
81	Decimal position

The range of the total field, specified in I_RNG of the field read IORB, may not exceed 80 characters. The range value can normally be computed with the following formula:

range = sum of R1...Rn subranges + number of auto-insert characters + 1 (if separate sign specified) + 1 (if decimal point specified).

Supervisory Message Processing

When a terminal is in field mode, the application may "escape" to a supervisory message line by issuing read/write orders with standard, non-extended IORBs. Escaping to the supervisory message line allows two-way communication between operator and application that does not disrupt the processing of a form displayed on the terminal. For example: An operator (who is using a terminal both for forms processing and as an operator console) receives a device unavailable message on the bottom line

of the terminal. The form being processed is not altered by the supervisory message. The operator acknowledges the supervisory message and continues processing the form.

IORB VALUES

Supervisory messages are designated by a common bit (bit 9) in the read/write device specific word. The use of this bit is optional in field mode, because supervisory message orders are already distinguished from field mode orders by being non-extended.

Bit 8 of I_DVS becomes significant when supervisory message writes are specified. If bit 8 = 0, supervisory messages must be acknowledged. If bit 8 = 1, acknowledgement by the operator is not required.

LOCATION OF MESSAGE LINE

If the terminal is defined at system building time as a VIP7800 or 7300 class terminal, the supervisory message line is the 25th line of the CRT. If the terminal is defined as a VIP7200 or 7207, the application may designate (in I_FCS) any line from 1 through 24 as the supervisory line.

PROCESSING ORDER

Supervisory message orders are processed by ATD in the order received, with write orders having priority over read orders. Assume for example, that four supervisory messages are issued and queued in the order listed: write, read, write, read. The two writes will be completed before the reads are processed.

If supervisory message orders are intermixed with extended IORB field mode orders, the messages are processed in the order received, with write orders again having priority over read orders. Assume, for example, that three orders are issued and queued in the order listed: field mode read, supervisory write, supervisory read. The orders will be processed in this order: supervisory write, field read, supervisory read.

SUPERVISORY MESSAGE CONVENTIONS

The following conventions apply to the processing of supervisory messages:

1. The receipt of a supervisory message by the LPH does not cause the premature termination of the current order, whether the current order is a supervisory message or normal field order.

2. Control byte and post-order control processing does not apply to supervisory messages.

3. If the type-ahead option was selected at connect time, a supervisory message results in a purge of the type-ahead character queue.

4. When writing a supervisory message, the application must not imbed in the message text control sequences that move the cursor (e.g., carriage return, line feed).

5. The range of a supervisory write order cannot exceed 80 characters. Data in excess of 80 characters is not sent to the terminal.

6. The operator must acknowledge the receipt of each supervisory message by depressing function key 10, the transmit key, or the CLEAR key.

7. The operator can edit a response to a supervisory message read through the use of TTY edit control characters.

8. The break function is not operational when a supervisory message read is being processed.

9. An operator keying in a response to a supervisory message read initiates transmission of the response by one of the following actions:

   a. Depressing the carriage return key

   b. Depressing the transmit key

   c. Entering the number of characters specified in I_RNG of the IORB issued by the application to read the operator's response.

10. The range of a supervisory message read order cannot exceed 80 bytes. If a longer range is specified, a range of 80 will be used.

11. ATD field mode applications that specify supervisory message processing and use the VIP7808, 7803, or 7303 in word processing mode must set to 1 bit 7 of I_DV2 in the connect IORB. This action ensures that the LPH keeps the terminal in word processing mode when servicing supervisory message requests.

Application Responsibilities in Processing Fields

The application is responsible for:

1. Initializing the read buffer with blanks, underscores, or semiconstant values.

2. Initializing the terminal display, through a field write order, with the same initialization sequence set in the read buffer.

3. Justification (left, right) after the field read is complete.

4. Decimal point alignment after the field read is complete.

5. Space suppression.

6. Logical validation of field content (beyond what is provided by ATD).

Field Mode Functions

Field mode supports six I/O request blocks:

Connect
Disconnect
Define Form
Read
Write
Break.

All but the break function require an extended-length IORB. When using an extended length IORB, bit 11 in I_CT1 must be set on, the right byte of I_EXT must specify a physical extension of seven words, and the left byte of I_EXT must specify a minimum total size of at least seven words.

CONNECT FUNCTION

An application selects field mode by using an extended-length connect IORB and setting bits 8, 9, 10, and 11 of I_DV2 to the field processing subfunction code of 2. (Bit 10 is set to one; the other three bits are zero.)

In field mode, the connect IORB can specify the following options.

Auto Call

Specification of auto call in I_DVS enables an application to establish a connection with either a 801-A or 801-C ACU data set. The auto call feature is described in Section 7.

Bell

The default setting of I_DVS allows the output of bells to a terminal. If the option is specified, the LPH suppresses the output of bells to a terminal even under error conditions. This means, for example, that the operator receives no indication when the LPH rejects entry into a field, or when entry of a terminator is required (when processing a must release field).

Validation Field Notification (VFN)

Specifying the VFN option (in I_DV2) causes the ATD, instead of issuing a bell, to post back the current read order with a return status of zero whenever the operator attempts to enter an invalid character into an active field.

Having specified the VFN option, the application determines the reason for the termination of the read order. If the order was terminated by the attempt to enter an invalid character (e.g., keying an "A" into a numeric subfield), ATD places an error code in I_CON. Having found this code, the application issues a supervisory message write to inform the operator of the error. Once the operator acknowledges the message and the supervisory message is posted back to the application, the application can reissue the interrupted field read and continue processing from the last valid keystroke (by means of a read with offset, which is described later in this section).

Selectable Field Validation Sets

This option (specified in I_DV2) allows the application to select the set of ASCII characters constituting a field type.

There are three validation sets that can be selected:

Standard ATD set
7700 set
7800 set

User applications must select the default ATD set. The other validation sets are used by system-supplied software that supports emulation of VIP7700 and 7804 terminals.

Word Processing Mode (WPM) Indicator

This option is specified (in I_DV2) by system-supplied software when the word processing graphics mode (WPM) of a VIP7803, 7808, or 7303 is to be used. This option is necessary to provide proper processing of supervisory messages when the terminal is in WPM mode.

Cursor Out of Field

If specified (in I_DV2), this option allows the operator to "cursor out" of a field and thus terminate the read of that field. The reason for termination is reported by ATD in the extended portion of the read IORB (I_TAB). If the option is not selected, the operator cannot use the cursor left key (at the beginning of a field) or cursor right key (at the end of a field) to terminate an active field read.

Type Ahead

This option, when specified (in I_DV2) helps to prevent the loss of input characters when a read order is not active (i.e., when a write order is active and/or a read order has not been issued by the application.)

If this option is chosen, ATD queues in a 32-character key-ahead buffer input characters that are keyed when a read order is not active. Later, when the read order becomes active, these characters are validated against the field attribute descriptor and echoed (if echo was requested). Detection of an invalid character causes an audible alarm to sound and the type-ahead character queue to be purged. Cursor right and left and end-of-field conditions are acted on by ATD when the read order becomes active.

If this option is not selected, characters are accepted only when a read order is currently active. The keying of characters when a read order is not active causes an audible alarm to sound.

The type-ahead queue is purged by any of the following events:

1. An input character in the queue is found to be invalid.
2. The application issues a supervisory message read or write order.
3. The operator presses the break key.
4. The terminal is disconnected.
5. The application issues a purge-all I/O order.
6. The application issues a read IORB with the the purge type-ahead queue bit set on.
7. The request issues a read order with terminal enquiry (ENQ) or with terminal read cursor address (RCA) specified as pre-order function in the IORB.

VIP7200, VIP7207 Supervisory Message Line

When issuing a connect to a VIP7200 or 7207, the application can specify (in the right byte of I_FCS) the line (row) to be used for supervisory messages. Possible values are hexadecimal 0 through 18. If zero is entered, line 24 is used. This field is ignored if the device is a VIP7800 or VIP7300 class terminal; in this case, line 25 is always used for supervisory messages.

Terminal Type (Device ID)

The application can check the device ID of the connected terminal by interrogating the right byte of I_QDP in the completed connect IORB.

Connect IORB (Field Mode)

This subsection summarizes the bit settings that govern the connect IORB options just described.

Bit Settings of I_DVS. Table 8-9 gives the signficance of bits in the connect IORB I_DVS word that are applicable to field mode ATD.

Table 8-9. ATD Word I_DVS in Connect IORB

Bit Number	Bit Value	Meaning for Connect Function
2	0	Do not use auto dial
	1	Use auto dial
3	0	Allow output of bells to the terminal
	1	Suppress output of bells to the terminal

Bits Setting of I_DV2. Table 8-10 gives the significance of bits of the connect IORB word I_DV2 that are applicable to field mode ATD.

Table 8-10. ATD Word I_DV2 in Connect IORB

Bit Number	Bit Value	Meaning for Connect Function
4	0	No Validation Field Notification (VFN) support
	1	VFN support
5,6	00	Use standard ATD field validation set (required setting)
	01	Use 7700 field validation set (reserved for system use)
	10	Use 7804 field validation set (reserved for system use)

Table 8-10 (cont). ATD Word I_DV2 in Connect IORB

Bit Number	Bit Value	Meaning for Connect Function
7	0	Terminal is VIP7200, 7207, 7801, 7301, 7307; or 7803, 7808, 7303 and is not operating in word processing graphics mode (required setting)
	1	Terminal is VIP7803, 7808, 7303 and is operating in word processing graphics mode (reserved for system use)
8		Must be 0*
9		Must be 0*
10		Must be 1*
11		Must be 0*
12	0	Operator not allowed to cursor out of field
	1	Operator allowed to cursor out of field (terminating field read)
13	0	No type ahead queue
	1	Type ahead queue is supported
*Bits 8 through 11 must be set as indicated to indicate a field mode connect.		

Bit Settings of I_FCS and I_ODP. The right byte of I_FCS specifies the line (or row) number of VIP7200 or 7207 that is used as the supervisory message line. Possible values are 0 through 18, hexadecimal. Zero indicates use of the 24th line (VIP7200 class terminals) or the 25th line (VIP7300 or 7800 class terminals).

Values Returned on Completion of a Connect Order. On completion of the connect order, the right byte of I_ODP contains the device ID of the terminal (refer to Table 8-3).

DISCONNECT FUNCTION (FIELD MODE)

The disconnect IORB is used to terminate field mode processing. A disconnect IORB can specify the following two options.

Abort Queued Orders

If this option is selected, all outstanding IORBs, even if active, are terminated with a device unavailable (010B) status. The disconnect order is then immediately serviced. If this option is not selected, all outstanding IORBs are allowed to complete (in the order of their issuance) before the disconnect order is serviced.

Hang Up

If this option is selected, the communications line is physically disconnected when the disconnect order is serviced. If this option is not selected, the terminal/line remains physically connected after processing of the disconnect order (i.e., the terminal is logically disconnected, but remains physically connected).

Disconnect IORB Word I_DVS. Table 8-11 gives the significance of bits of the disconnect IORB I_DVS word that are applicable to the disconnect options just described.

Table 8-11. ATD Word I_DVS in Disconnect IORB

Bit Number	Bit Value	Meaning for Disconnect Function
14	0	Abort outstanding requests
	1	Wait until outstanding requests complete before disconnecting the terminal
15	0	Hang up the phone
	1	Do not hang up phone

READ FUNCTION (FIELD MODE)

An extended-length field read IORB is used to obtain validated input that has been keyed into a field displayed at a terminal. The input to a field is validated by means of a field descriptor, which must be associated with the field read order. The descriptor may either be integrated into the read IORB or belong to a table of descriptors pointed to by a define form IORB. (For further detail, see "Field Descriptor and Define Form" earlier in this section).

Pre-order Control

Pre-order control arguments are specified in I_DV2 and I_CON of the read order IORB. Pre-order control is used to perform the following actions prior to a field read:

- Positon cursor
- Issue bell
- Erase line (i.e., clear screen from cursor position to end of line)
- Issue enquiry command (ENQ) to VIP7801, 7803, 7808, 7301, 7307 and read terminal's response
- Issue a read cursor request command (RCA) to terminal and read current position of the cursor.

ENQ and RCA Commands. If the pre-order control request is an ENQ or RCA command, the read is not treated as a field read. After sending an ENQ or RCA control sequence to the terminal, the LPH places the terminal's response in the buffer associated with the read request.

Before issuing an ENQ or RCA read order, the application must specify in the read IORB no echo of incoming characters and no post-order control.

(For more information about the ENQ and RCA commands, see the hardware documentation of the terminal in question.)

Termination of a Field Read

An operator intentionally terminates a field read by one of two actions:

1. The operator types a valid data character into the last position of a field that is not a must release field. This action sets I_RNG to and I_TAB to zero.
2. The operator types a control sequence that terminates the read order. The character(s) making up the control sequence must fall in certain ranges (defined below) of the ASCII character set; the significance of the sequence, however, is determined by the application. Keying a control sequence sets a non-zero residual range in I_RSR. The terminator sequence is stored in the I_TAB and I_CON fields of the IORB, and not the buffer; it is not included in the residual range calculation or echoed to the terminal.

Terminating Sequences. The terminating sequence may be a one-character control character or an escape sequence from one to four characters long.

1. One-Character Terminating Codes. A One-character terminator must be one of the following ASCII codes: 00-1A, 1C-1F, 7F. Note that codes 10 and 11 are not treated as terminators by ATD if the terminal is a VIP7207 or 7307. The use of codes 10 and 11 is not recommended if compatibility with all terminal types is desired.

2. Multi-character Terminating Codes. Multi-character terminators are two-, three-, or four-character sequences beginning with the escape code 1B. The second character of the sequence must be in the range 20-7E.

   A two-character sequence must consist of the escape character (1B) followed by 20 to 57; 59 to 5A; 5C to 72; or 74 to 7E.

   The VIP7800 and 7300 terminal classes support three- and four-character escape sequences. The first two characters must be 1B followed by one of the following: 58, 5B, or 73.

Escape sequences longer than four characters are not supported; the fifth and any successive character(s) are treated as data.

For further information on escape sequences, refer to documentation describing a specific terminal.

ATD Handling of Termination Codes

The terminating sequence keyed by an operator is placed by ATD in extended IORB fields I_TAB and I_CON. The following rules apply.

1. One-Character codes. One-character codes are placed in the right byte of I_TAB and are in the range of 00 - 1A, 1C - 1F or 7F.

2. Two-character escape sequences. The escape character (1B) is not stored. The second character is stored in the right byte of I_TAB and is in the range of 20 - 7E (excluding 5B, 58, and 73).

3. Three- and four-character escape sequences. The escape character (1B) is not stored. The second character is stored in the right byte of I_TAB and is 5B, 58, or 73. The left byte of I_CON contains the third character; the right byte of I_CON contains the fourth character. The value X'00' in the right byte of I_CON signifies that the terminating code is a three-character escape sequence.

Entry of Invalid Characters

The effect of entering an invalid character into a field requiring validation depends on whether the validation failure notification (VFN) option was selected at connect time.

1. VFN Option Not Selected. An audible alarm sounds (if output of bells is supported), the cursor remains in its current position, and the invalid character is not echoed. The LPH continues to process the current order without notifying the application of the input error.

2. VFN Option Selected. Field read order is returned to application with a 0 status in I_CTl. Right byte of I_TAB contains X'FF', indicating that I_CON contains one of the following error codes:

   1 = Illegal entry into a digit subfield
   2 = Illegal entry into a numeric subfield
   3 = Illegal entry into an alphabetic subfield
   4 = Illegal entry into an alphanumeric subfield

Residual Range and Relative Residual Range

When a field read order is terminated, the residual range, returned in I_RSR of the IORB, reflects the maximum cursor position reached while the read order was active. The relative residual range, returned in I_QDP of the IORB, reflects the position of the cursor when the order was terminated. The values of residual and relative residual range may differ if the cursor back (<-) key was entered during a read order. Suppose, for example, that the operator keys

AXC<-<-B<-

followed by a carriage return. The residual range shows that three characters (ABC) were entered; the relative residual range indicates that the cursor was in the second position when the order was terminated by a carriage return.

The residual and relative residual range are set equal to the original range if a read order is prematurely terminated by depression of the break key, a communication line loss, a purge-all, or an abortive disconnect.

Use of Cursor Keys

When the operator moves the cursor left (<-) or right (->) within a field, the LPH's buffer pointer is adjusted and the buffer contents remain unchanged. For example, after the operator keys

ABC->->

followed by a carriage return, the buffer contains ABCxx, with xx being the previous contents of the buffer. (The residual range indicates that five characters were entered.)

Statistics

The total keystroke count for a field read is returned in I_FCS of the field read IORB when the order terminates. When the read order is active, the count is incremented once for each of the following:

- Data character (valid or invalid)
- Cursor right (->)
- Cursor left (<-)
- The terminating character sequence.

Statistics are not returned in the IORB if the read order is prematurely terminated (e.g., by a communication line loss).

Read With Offset

An application can specify an offset when issuing a field read order so that the operator can start entering data in the middle of a field.

When issuing a read with offset, an application does the following:

- Specifies in I_BAD the starting address of a buffer that contains the data from the field previously read.
- Specifies in I_RNG the size of the buffer pointed to by I_BAD.
- Specifies in I_HDR the offset from the start of a field to a position within the field where the cursor is to be placed and where the read with offset is to begin. Permissible values are in the range 1 through 4F, hexadecimal.
- Optionally, specifies in I_CON the cursor position to the start of the field.

The following example shows the procedure and purpose of issuing a read with offset.

Example:

A 20-character alphabetic field begins in row 2 column 1. The application previously issued a field read with no offset, but the field entered by the operator contained an invalid character in the tenth position of the field. The application recognizes the error and reissues the read with:

- A pre-order bell
- A pre-order positioning of the cursor at row 2, column 1
- An offset of 9 specified in I_HDR
- The address and range of the buffer containing the previously read data, specified in I_BAD and I_RNG, respectively.

During a read with offset, the operator is allowed to cursor left or right within the entire field. Cursoring out of a field follows the normal termination rules.

The LPH calculates the residual range and the relative residual range for a read with offset order as if the operator had entered the characters preceding the specified offset.

Read with offset may be used with or without the type-ahead option.

Type-Ahead

If the type-ahead option was selected at connect time, the application may select the "purge type-ahead queue" option in the field read IORB. This option causes the LPH to purge the type-ahead queue before processing the read order. The option is useful if the application detects an error in field read and wants to re-issue the read after purging the queue.

Cursor Out of Field

When issuing a field read order, an application can override the selection of the cursor-out-of-field option made at connect time. That is, by setting a bit in I_DV2 of the read IORB, the application can specify that the operator cannot cursor out of the field.

Support of VIP7207 and 7307 Terminals

Through support of the ALPHA key and implied numeric shift, ATD supports data entry operations on the VIP7207 and 7307 terminals.

Alpha Key Functionality. The purpose of this functionality is to allow the operator to enter alpha (i.e., lower case) characters from a data entry terminal while the terminal is shifted to uppercase as a result of numeric lock or implied numeric shift. It is perceived by the operator as a terminal function related to character entry, and is not tied into the field validation operation. Field validation checks are done after the character is translated; if the resultant character is invalid it is rejected at that time.

The data entry terminal transmits a code 10 when the ALPHA key is depressed and a code 11 when the key is released. The LPH interprets code 10 as a shift to the "alpha" set of characters, translating the data characters following the code 10 into equivalent alpha codes until a code 11 is received. ATD so interprets codes 10 and 11 whether or not type-ahead is in effect.

When the operator is responding to a supervisory message read, ATD treats codes 10 and 11 as data, placing them in the application's buffer; no translation is performed. After the supervisory message read is complete, the LPH reverts to the mode (ALPHA or implied numeric shift) that was in effect immediately before the supervisory read.

Numeric Shift Functionality. The purpose of this option (specified in I_DV2) is to reduce the number of keystrokes required of the operator during the entry of numeric data by enabling the application to shift the state of the terminal instead of requiring the operator to depress the numeric shift key. The use of this option is not restricted to numeric type validation fields, and it can be used wherever it will save the operator keystrokes. Thus, for alphanumeric fields that typically consist mostly of digits, this implied shift would cause the terminal to echo digits and the ALPHA key could be used to enter the occasional letter.

When a field read with implied numeric shift is requested, the characters entered are translated before the field validation operation is performed.

The following is an example of the use of the implied numeric shift option:

> The operator normally enters alpha data. The terminal is set for alpha: the numeric lock is not set and the ALPHA key is not used. The application issues an order to read a three-character alphabetic subfield and a three-character digit subfield with the implied numeric shift option. The operator, using the central keyboard (not the numeric keypad), enters ABCUIO. ABC123 is placed in the application buffer.

Table 8-12 lists the data codes produced by a key in its unshifted (alpha) state and shifted (numeric) state. The two characters shown in each line of the table are produced by a single key. The first character is produced when the keyboard is unshifted or when the alpha key functionality is in effect. The second character is generated when the key board is shifted or when the implied numeric shift option is in effect.

Table 8-12. Data Entry Keyboard Unshifted/Shifted Translations

Unshifted (alpha)	Shifted (numeric)	Terminal
S (53)	> (3E)	
X (58)	? (3F)	
T (54)	[(5B)	VIP7207
R (52)	[(5B)	VIP7307
H (48)	\ (5C)	VIP7207
G (47)	] (5D)	VIP7207
T (54)	] (5D)	VIP7307
R (52)	^ (5E)	VIP7207
W (57)	_ (5F)	
{ (7B)	| (7C)	VIP7207
{ (7B)	} (7D)	VIP7307
} (7D)	~ (7E)	VIP7207
^ (5E)	~ (7E)	VIP7307
G (47)	none	VIP7303
B (42)	! (21)	
C (43)	" (22)	
@ (40)	# (23)	
* (2A)	$ (24)	
P (50)	& (26)	
N (4E)	((28)	
E (45)	) (29)	
Q (51)	+ (2B)	
% (25)	, (2C)	
< (3C)	. (2E)	
none	/ (2F)	VIP7207
H (48)	/ (2F)	VIP7307
/ (2F)	0 (30)	
U (55)	1 (31)	
I (49)	2 (32)	
O (4F)	3 (33)	
J (4A)	4 (34)	
K (4B)	5 (35)	
L (4C)	6 (36)	
M (4D)	7 (37)	
, (2C)	8 (38)	
. (2E)	9 (39)	
D (44)	: (3A)	
F (46)	; (3B)	
V (56)	= (3D)	

NOTES

1. "None" means that no code is generated.
2. Unless specified, the code translations apply to both VIP7207 and 7307 terminals.
3. Keys not represented in the table generate the same code in unshifted or shifted state.

Read IORB (Field Mode)

This subsection summarizes the bit settings that govern the field read IORB options just described.

Bit Settings of I_DVS. Table 8-13 gives the significance of bits of the field read IORB I_DVS word that are applicable to field mode ATD.

Table 8-13. ATD Word I_DVS in Field Read IORB

Bit Number	Bit Value	Meaning for Field Read Function
10	0 1	Do not echo input Echo input

Bit Settings of I_DV2. Table 8-14 shows bits in the field read IORB word I_DV2 that are significant to ATD.

Bit Settings in I_CON. This field can be used to specify pre-order control. If so used, bit 14 of I_DV2 must be set. I_CON can be used to specify two kinds of pre-order control

1. Pre-order cursor positioning. The application must indicate this use of I_CON by setting bit 15 of I_DV2 to one.

2. Pre-order control other than cursor positioning. The application must indicate this use of I_CON by setting bit 15 of I_DV2 to zero.

Table 8-14. ATD Word I_DV2 in Field Read IORB

Bit Number	Bit Value	Meaning for Field Read Function
0	0	Do not purge type-ahead queue
	1	Purge type-ahead queue
1	0	No implied numeric shift
	1	Implied numeric shift
2	0	No integrated field descriptor (define-form order required)
	1	Integrated field descriptor (starting at I_LOG)
7	0	Do not override cursor-out-of field capability
	1	Override cursor-out-of field capability specified in connect I_DV2 (field read terminates when attempt is made to cursor-out of a field)
12	0	Do not send pre-order bell
	1	Send pre-order bell
13	0	Do not send pre-order erase-line escape sequence
	1	Send pre-order erase-line escape sequence
14	0	I_CON is not meaningful (no pre-order control)
	1	I_CON contains pre-order control information
15	0	Right byte of I_CON contains pre-order control (see Table 8-15); left byte must be zero
	1	I_CON contains pre-order cursor positioning information (see Table 8-15)

Table 8-15 shows the values of I_CON when used for either kind of pre-order control.

Table 8-15. ATD Word I_CON in Field Read IORB.

Bit Number	Hex Value	Meaning for Field Read Function
Pre-Order Control Cursor Positioning Information		
0-7	01-50	Defines column coordinate (hexadecimal)
8-15	01-18	Defines row coordinate (hexadecimal)
Other Pre-Order Control Information		
0-7		Must be zero
8-15	00	Line feed and carriage return
	01	Line feed
	02	Carriage return
	03	Bell
	04	Reserved for future use
	05	Reserved for future use
	06	Restore device's default attributes (VIP7800, 7300 class terminals); high intensity (VIP7200 class terminals)
	07	Low intensity attribute (VIP7800, 7300, 7200 class terminals)
	08	Cursor up
	09	Cursor down
	0A	Cursor forward
	0B	Cursor back
	0C	Cursor home
	0D	Erase end of line

Table 8-15 (cont). ATD Word I_CON in Field Read IORB.

Bit Number	Hex Value	Meaning for Field Read Function
Other Pre-Order Control Information (cont.)		
	OE	Erase end of display
	0F	Clear (VIP7800, 7300 class terminals); reset (VIP7200 class terminals)
	10	Read cursor request binary (VIP7800, 7300 class terminals); read cursor address (VIP7200 class terminals)
	11	Blink (VIP7800, 7300 class terminals)
	12	Hide (VIP7800, 7300 class terminals)
	13	Inverse video (VIP7800, 7300 class terminals)
	14	Underline (VIP7800, 7300 class terminals)
	15	Secondary character set (VIP7800 class terminals)
	16	Enquiry (VIP7800, 7300 class terminals)

NOTES

1. If codes 11 through 16 are used for terminal classes other than VIP7800 and 7300, an invalid parameter error (0104) will be returned.

2. When specifying codes 10 or 16, the application must supply, in IORB fields I_ADR and I_RNG respectively, the address and size of a buffer to receive the terminal's response. A buffer size of 4 bytes is required if code 10 is specified; a buffer size of 9 is required if code 16 is specified.

Bit Settings in I_HDR and I_TAB. If an application issues a read with offset, the right byte of I_HDR must contain the byte offset, expressed as a hexadecimal value in the range 1 through 4F. If no offset is required, the right byte of I_HDR must be zero.

When an a field read order is issued in conjunction with a define form order, I_TAB contains a word offset to the proper field attribute descriptor.

Values Returned by a Field Read Order

The following paragraphs summarize the information returned by ATD in fields of a terminated field read IORB.

I_RSR shows the maximum cursor position (offset) upon termination of the field read order.

I_FCS shows the total number keystrokes entered by the operator during the field read.

I_ST2 indicates, by a value of one in bit 15, that validated data was entered into the field.

I_QDP shows the current cursor position (offset) upon termination of the field read order.

I_TAB indicates termination condition as shown below:

0 = End of range. Valid data has been entered into the entire field.

-1 = Invalid character entered into field. If the VFN option was selected at connect time, I_CON provides additional information (see "Entry of Invalid Characters into a Field" earlier in this section.)

>0 = ASCII code for one of the following:

Single character terminator entered by operator

Second character of a two-character escape sequence

Second character of a three- or four-character escape sequence; I_CON contains remaining character(s) of the sequence

For the permissible values of terminator characters and sequences see "Termination of Field Read" earlier in this section.

WRITE FUNCTION (FIELD MODE)

An extended-length IORB is used for all write orders directed against a terminal connected in field mode.

Write orders are typically used to:

- Display on the terminal screen a set of field "templates" associated with a form
- Purge all outstanding field read and write orders.

Purge All Subfunction

The purge all option is a special form of the field write order. It is exercised by specifying a subfunction code of three in I_DV2, bits 8 through 11. Bits 10 and 11 are one; the other bits are zero. When this subfunction is specified, all other bit settings in I_DV2 and I_DVS are ignored. The write order causes outstanding read and/or write orders (active and queued) to be posted with a device unavailable (010B) status. Further, if the type-ahead option was specified at connect time, the type-ahead queue is purged.

Quit on Break Option

If this option is specified (in I_DVS), a break signal can prematurely terminate an active write order.

Pre-order Control

Four bits in I_DV2 control pre-order activity. By setting the range I_RNG to zero, the application can issue a write order that requests only pre-order activity. Alternatively, the write order can request both pre-order activity and the output of data to the terminal. In either case, the subfunction code (bits 8 through 11) of I_DV2 must be zero.

By manipulating bits in I_DV2, an application can:

- Send a bell
- Erase end-of line
- Use I_CON for cursor positioning operations
- Use I_CON for pre-order control operations.

These options are also available with field read orders and have been described in earlier parts of this section that concern the field read function.

Write IORB (Field Mode)

This subsection describes bit settings in the field write IORB that govern the options just described.

Bit Settings in I_DVS. Table 8-16 gives the significance of bits of the IORB word I_DVS that are applicable to field mode write.

Table 8-16. ATD Word I_DVS in Field Write IORB.

Bit Number	Bit Value	Meaning for Field Write Function
7	0	Stop output on detecting break
	1	Do not stop output on detecting break

Bit Settings in I_DV2. Table 8-17 gives the significance of bits in IORB word I_DV2 that are applicable to field mode write.

Table 8-17. ATD Word I_DV2 in Field Write IORB.

Bit Number	Bit Value	Meaning for Field Write Function
8	0	Must be zero
9	0	Must be zero
10-11	00	Normal write
	11	Purge all outstanding read and/or write orders
12	0	Do not send pre-order bell
	1	Send pre-order bell
13	0	Do not send pre-order erase-line
	1	Send pre-order erase-line
14	0	I_CON is not meaningful (no pre-order control)
	1	I_CON contains pre-order control information

Table 8-17 (cont). ATD Word I_DV2 in Field Write IORB.

Bit Number	Bit Value	Meaning for Field Write Function
15	0	Right byte of I_CON contains pre-order control (see Table 8-15); left byte must be zero
	1	I_CON contains pre-order cursor positioning (see Table 8 -15)

Bit Settings in I_CON. The bit settings in this field is the same as those previously described for the field read function, with this exception: In a field write IORB, I_CON does not support codes 10 (read cursor address) and 16 (Enquiry).

FIELD MODE DEVICE CONFIGURATION

Hardware switches on a device connected in field mode should be set in the following positions. (The device may not support all of the switches mentioned below).

CHARACTER/BUFFER switch in CHARACTER position
DUPLEX HALF/FULL switch in FULL position
LOCAL COPY/ECHO switch in ECHO position
ROLL/NO ROLL switch in NO ROLL position
Speed set between 1200 and 9600 bits per second

FIELD MODE RETURN STATUS CODES

The following return status codes are returned in the R1 register. The status code returned in I_CT1 is the right byte of the status code returned in the R1 register when the I/O order is complete.

Invalid Argument Status (0104)

This status is returned for the following reasons:

- In a field read IORB
  - I_RNG (buffer size) is zero
  - Invalid pre-order control option or cursor position coordinate in I_CON
  - The format or values of a field descriptor are invalid.

- In a field write IORB

  - Invalid pre-order control option or cursor position coordinates in I_CON

  - Improper bit settings in I_DV2 (bits 8 through 12 must be all zero; else bits 8 and 9 set to zero and bits 11 and 12 set to one).

Inconsistent Request Status (010C)

This status is returned for the following reasons:

- In a field connect IORB

  - The IORB specifies a field mode connect to a terminal that is supporting (a connected) serial printer that is attached to the terminal by a buffered printer adapter.

- In a define form IORB

  - A read order is presently using an outstanding and active define form order; definition of a new define form is not allowed.

- In a field read IORB

  - A field attribute descriptor has not been specified.

FIELD MCDE ERROR PROCESSING

When a parity error is detected in keystroke input, an audible alarm sounds and the typed character is ignored. When the read order is posted, the return status in I_ST indicates detection of parity error(s) (bit 9 = 1).

If a framing error or receive overrun condition is detected, the read order terminates and a hardware error (0107) is returned; I_ST indicates the specific reason for abnormal termination.

FIELD MODE TIMEOUT PROCESSING

Timeouts may occur during the processing of read orders. A timeout occurs when the operator does not terminate the input within 5 minutes after entering the first character. There is no timeout if the operator does not enter any characters.

Write orders do not incur timeouts.

BLOCK MODE

Block mode is applicable only to the VIP7800 class of terminals (VIP7801, 7803, and 7808) and is intended to support the terminal in its native text or forms mode.

Block mode supports five functions:

- Connect
- Disconnect
- Read
- Write
- Break.

These functions are requested through standard-length IORBs. An application can optionally use an extended IORB for a connect operation.

Connect Function

A connect order establishes the mode in which the connected terminal operates. Block mode is selected by setting bit 0 of I_DVS to one.

If an extended-length connect IORB is used, the terminal's device ID is returned in the IORB extension (right byte of field I_QDP).

An application specifies in I_RNG of the connect IORB the size of data blocks to be transmitted from the terminal. Permissible block sizes range from 22 to 270F bytes, hexadecimal. If an application fails to specify a valid block size, the connect order is rejected with an 0104 (invalid argument) error code.

Transmitted blocks terminate with either an end-of-block (ETB) or an end-of-text (ETX). When a block is transmitted from the terminal, the type of terminator (ETB or ETX) is passed to the application through the IORB and through an optional control word, which is described below.

The following options can be specified when connecting in block mode.

AUTO CALL

The Auto Call option, which is supported by all system-supplied LPHs, is described in Section 7. This option allows an application to establish a connection using an 801-A or an 801-C ACU data set.

CONTROL WORD

At connect time, an application can specify control word processing for subsequent read and write orders. If this option is specified, ATD treats the first two bytes of the user's buffer as a control word. If control byte processing is also specified, the third byte of the user's buffer is considered the control byte.

ATD uses the control word primarily to pass information to an application on completion of read orders. ATD places similar information in the IORB word I_ST when a read order completes.

The first byte of the control word contains information that is passed to the application upon completion of a read order. It has the following format:

Bit 3

0 = ETX terminated block
1 = ETB terminated block

Bit 5

0 = -
1 = Long block; data lost

Bit 6

0 = -
1 = Block missed; data lost

The second byte of the control word specifies the logical resource number of the referenced terminal.

If specified, the control word must be included in the range (I_RNG) of the associated data buffer.

SPACE SUPPRESSION

If this option is specified, ATD configures the terminal to suppress spaces, in certain instances, when transmitting data. One example of space suppression is the replacement of spaces between fields by a horizontal tab character; another example is the elimination of spaces at the end of lines that are terminated by a carriage return and line feed. For additional details, consult the documentation for the terminal in question.

NO ROLL

Selecting this option keeps the terminal from scrolling line 1 "off the screen" when text (including a carriage return) is entered into line 24. This option is especially useful to applications that process forms.

If this option is not specified, the screen scrolls as new text is entered in line 24. Roll mode is the customary operating mode chosen by an application that processes line-at-a-time input from the terminal.

Connect IORB (Block Mode)

This subsection summarizes the bit settings that govern the connect options already described.

BIT SETTINGS OF I_DVS

Table 8-18 gives the significance of bits of the connect I_DVS word that are applicable to block mode.

WORD I_RNG

A block size must be specified in this field if block mode is selected (bit 0 of I_DVS is one).

Table 8-18. I_DVS Word in Connect IORB (Block Mode)

Bit Number	Bit Value	Meaning for Connect Function
0	0	Do not use block mode
	1	Use block mode
2	0	Do not use auto dial
	1	Use auto dial
4	0	Include control word
	1	Do not include control word
8	0	Do not use space suppression
	1	Use space suppression
9	0	Use roll
	1	Use no roll

WORD I_ST

This field is significant when a serial printer is attached to the terminal by means of a VIP7800 buffered printer adapter (VAF7821). On connect orders, the field specifies whether the terminal or attached printer is being addressed. The permitted values are:

0 = Terminal
1 = Attached serial printer

WORD I_QDP

Upon completion of a connect order, ATD returns in the right byte of I_QDP the device ID of the terminal (refer to Table 8-3).

Disconnect Function (Block Mode)

An application uses the disconnect IORB to terminate block mode processing.

The following paragraphs describe the options that an application can specify with a disconnect order.

ABORT QUEUED ORDERS

If the abort option is specified, outstanding IORBs (active and queued) are terminated with a "device unavailable" status (010B). The disconnect order is immediately serviced. If the abort option is not specified, all outstanding IORBs are allowed to complete before the disconnect order is serviced.

HANG UP

If the hang-up option is selected, the terminal is physically disconnected when the disconnect order is serviced. If the hang-up option is not specified, the communications connection remains active after servicing of the disconnect order (i.e., the terminal is logically disconnected but remains physically connected).

Disconnect IORB (Block Mode)

This subsection summarizes the IORB bit settings that govern the disconnect options just described.

BIT SETTINGS OF I_DVS

Table 8-19 shows bits of the disconnect IORB that are applicable to the block mode of ATD.

Table 8-19. I_DVS Word in Disconnect IORB (Block Mode)

Bit Number	Bit Value	Meaning for Disconnect Function
14	0	Abort outstanding requests
	1	Wait until outstanding requests complete before disconnecting the terminal
15	0	Hang up the phone
	1	Do not hang up the phone

BIT SETTING IN WORD I_ST

This field is significant when a serial printer is attached to the terminal by means of a VIP7800 buffered printer adapter (VAF782l). On disconnect orders, the field specifies whether the terminal or attached printer is being addressed. The permitted values are:

0 = Terminal
1 = Attached serial printer

Read Function (Block Mode)

The read order is used to obtain blocks of data transmitted from the terminal. It is the application's responsibility to specify a buffer size large enough to hold a complete block of data. If a block of data exceeds the buffer capacity of the order, the IORB is posted with a "long record" status (bit 6 of I_ST is 1).

OPERATOR FUNCTIONS

The operator edits information at the terminal by using the following keys:

- Cursor control
- Character insertion/deletion
- Line insertion/deletion
- Line/screen erase.

The operator signals termination of input by pressing the TRANSMIT key. A break key enables the operator to interrupt a read order or to (possibly) terminate a write order.

APPLICATION FUNCTIONS

An application selects the following options by setting bits in the device-specific word (I_DVS) of the IORB.

Abort Read

If this option is specified, ATD posts to the application any active and queued read IORBs. The posted IORBs show a device unavailable status (010B) in I_CTl and the abort indicator (bit 0) in I_ST set to one. The read order issued with this option causes no I/O activity; it is posted back to the application with a zero status.

Supervisory Messages

Specification of this option indicates that the read order is directed to the supevisory message line. This option is meaningful only if the terminal is operating in no-roll mode. In no-roll mode, the supervisory message line is line 25. In roll mode, supervisory message reads are treated as normal reads.

Line Feed and Carriage Return

Specifying the line feed and/or carriage return option causes, respectively, a line feed and/or carriage return to be sent to the terminal when the read order is completed.

READ IORB (BLOCK MODE)

An application specifies the options just described by setting bits in the IORB word I_DVS. Table 8-20 gives the significance of these bits.

Table 8-20. ATD Word I_DVS in Block Mode Read IORB

Bit Number	Bit Value	Meaning for Block Read Function
0	0	Normal read
	1	Abort read
9	0	Normal read
	1	Supervisory message read
11	0	Do not send post-order line feed
	1	Send post-order line feed
12	0	Send post-order carriage return
	1	Do not send post-order carriage return

Write Function (Block Mode)

The write order is used to transmit data blocks to the terminal.

WRITE ORDER PROCESSING

Write orders have priority over read orders. If a read order has been issued but is not in progress, any issued write order executes immediately. Once all outstanding write orders have completed, the outstanding read order is reestablished. If a read order is in progress (i.e., entry of data from the terminal has begun), the write order waits for the read to complete.

KEYBOARD LOCK

Before the write order is executed by ATD, the LPH locks the terminal's keyboard. This action prevents processing conflict between the LPH and terminal. After the write order is processed, the keyboard is unlocked if the completed write order specified an ETX terminator (indicating the end of the message transmission to the terminal). If, however, the contents of the write order contains an escape sequence that elicits a response from the terminal, the device will ignore the keyboard unlock command; the application must issue another write order to unlock the keyboard.

WRITE ORDER OPTIONS

An application can specify the following options in I_DVS of the write IORB.

Abort Write

If this option is specified, ATD posts to the application any active and queued write IORBs. The posted IORBs show a device unavailable status (010B) in I_CTl and the abort indicator (bit 0) in I_ST set to one. A write order issued with this option causes no I/O activity; it is posted back to the application with a zero status.

Preemptive Data Write

This option is meaningful only when the terminal is actively transmitting data. The option allows a write order to be processed between the transmission (by the terminal) of two ETB blocks or one ETB block followed by an ETX block. Normally, once a read operation is started by the application (to receive terminal transmissions), it is allowed to proceed (often requiring the issuance of several read IORBs) until the last text block (terminated by ETX) is received.

Control Byte Processing

Specification of control byte processing indicates that the first byte in the application's output buffer is to be used for pre-order control. A control byte must be included in the range (I_RNG) of data to be written to the terminal. For a detailed description of this option, including control byte format, refer to "Control Byte Processing" earlier in this section.

ETX/ETB Option

As mentioned earlier, ATD locks the keyboard during processing of a block mode write order. If the write order specifies ETB, indicating that a another block of the message is to follow, the keyboard remains locked after completion of the write order. Alternatively, if the write order specifies ETX, indicating the end of the message, the keyboard unlocks after completion of the order.

Quit On Break

If this option is specified in I_DVS, a break signal can interrupt the execution of an active write order. Otherwise, a break signal cannot be used to terminate an active write order prematurely.

Supervisory Messages

Specification of this option indicates that the write order is directed to the supervisory message line. This option is meaningful only if the terminal is operating in no-roll mode. In no-roll mode, the supervisory message line is line 25. In roll mode, supervisory message writes are treated as normal writes.

Supervisory Message Acknowledgement

If this option is specified, it indicates that a supervisory message written to a terminal is to be acknowledged by the terminal operator. Again, supervisory messages are meaningful only if the terminal has been connected in no roll mode. In roll mode, supervisory messages are treated as normal writes and the acknowledgement option does not apply. For a full discussion of this topic, refer to "Supervisory Message Processing" earlier in this section.

Line Feed and Carriage Return

Specifying the line feed and/or carriage return option causes, respectively, a line feed and/or carriage return to be sent to the terminal when the write order is completed.

Write IORB (Block Mode)

This subsection summarizes the bit settings that govern the write order options already described.

BIT SETTINGS OF I_DVS

Table 8-21 gives the significance of bits of the write I_DVS word that are applicable to block mode.

Table 8-21. ATD Word I_DVS in Block Mode Write IORB

Bit Number	Bit Value	Meaning for Block Write Function
0	0	Normal write
	1	Abort write
3	0	Normal write
	1	Preemptive write
4	0	Include control byte
	1	Do not include control byte
6	0	ETX (unlock keyboard after write order completes)
	1	ETB (keep keyboard locked after write order completes)
7	0	Stop output on detection of a break
	1	Do not stop output on detection of a break
8	0	Operator must acknowledge supervisory message
	1	Operator need not acknowledge supervisory message
9	0	Normal write
	1	Supervisory message write
11	0	Do not send post-order line feed
	1	Send post-order line feed
12	0	Send post-order carriage return
	1	Do not send post-order carriage return

BIT SETTING IN WORD I_ST

This field is significant when a serial printer is attached to the terminal by means of a VIP7800 buffered printer adapter (VAF7821). On write orders, the field specifies whether the terminal or printer is being addressed. The permitted values are:

0 = Terminal
1 = Attached serial printer

Device Configuration (Block Mode)

In block mode, the speed of a terminal must be configured between 110 and 9600 bits per second.

Return Status Codes (Block Mode)

ATD returns status codes in I_CT1 and I_ST. The status code returned in I_CT1 is the right byte of the status returned in the R1 register (when the I/O order is completed).

STATUS CODES IN I_CT1

The invalid argument status (0104) is returned when an invalid block size is specified in I_RNG of a connect IORB.

The device unavailable status (010B) is returned when a read or write order is purged as a result of a purge-all read request or purge-all write request, respectively.

The inconsistent request status (010C) is returned for a read order that is issued subsequent to a data loss. This status indicates that one or more data blocks were missed prior to the issuance of the current read order.

STATUS CODES IN I_ST

Table 8-22 shows status information returned in I_ST upon completion of a block mode order.

Table 8-22. IORB Word I_ST (Block Mode)

Bit	Meaning when Bit Set to One
0	Read or write order aborted
1	ETB received; (ETX received if bit off)
3	Block missed; was received from terminal without a read order having been issued
6	Long record received; buffer insufficient to contain received data

Error Processing (Block Mode)

When a parity error is detected on a data transmission from the terminal, an ASCII SUB character (1A) is placed in the application's buffer in lieu of the erroneous character. The read order is posted with a hardware error status (0107), and bit 9 of I_ST is set to one to indicate that one or more parity errors were detected during the read.

Detection of a framing error or receive overrun condition prematurely terminates the read order. The order is posted with a hardware error status (0107); I_ST indicates the reason for abnormal termination.

Timeout Processing (Block Mode)

In block mode, there are no timeouts for read or write orders.

ROP MODE

The ROP (receive-only printer) mode of ATD services selected serial printers that use an ETX/ACK protocol. It supports four functions, using standard-length IORBs:

Connect
Disconnect
Write
Read

ETX/ACK Protocol

Use of this protocol avoids a buffer overflow condition, in which an application transmits data to a device faster than the device can print the data. Buffer overflow is most likely to occur while the device is executing commands, such as carriage return or form feed, that move the print head or carriage. Without an ETX/ACK protocol, the application or device driver must pad data transmissions with fill characters, which the device does not print. While the fill characters are being edited out, the device has time to perform carriage returns or line feeds.

The ETX/ACK protocol renders padding unnecessary. Using this protocol, the LPH sends data to the printer a block or frame at a time. (The size of the block or frame depends on the buffering capacity of the device.) The LPH terminates the block with the ETX character. The serial printer responds with an ACK control character when (if the unit is double-buffered) it can accept another block or when it has sucessfully printed the last block of data. Having received the ACK control character, the LPH starts transmitting the next data block.

The ROP LPH supports a basic and advanced type of ETX/ACK protocol.

BASIC ETX/ACK PROTOCOL

The basic ETX/ACK is used by letter-quality serial printers (the PRJ1004 and 7007). It supports:

- A basic transmission procedure
- Detection of off-line serial printer conditions by means of an attention read order
- Report of the printer's marketing identifier by means of a status read order.

ADVANCED ETX/ACK PROTOCOL

The advanced ETX/ACK is used by the PRU7070 and 7075 serial printers. It supports:

- An advanced ETX/ACK transmission procedure called the asynchronous serial printer interface (ASPI)
- Detection and report of all off-line serial printer conditions
- Report of the printer's marketing identifier and device status by means of a status read order.

Connect Function

An application selects the ROP mode of ATD by setting bit 10 of I_DVS to one when issuing the connect order.

When the device connected is a PRU7070 or 7075, the LPH issues an enquiry to the device for status. The serial printer's response to the request for status allows ATD to specialize its processing to the characteristics of the device. If the device fails to respond to the request for status, ATD posts back the connect order with a device unavailable (010B) status.

AUTO CALL

When connecting in ROP mode, the application can specify the auto call option. Specifying auto call in I_DVS enables an application to establish a connection using either an 801-A or an 801-C ACU data set.

Connect IORB (ROP Mode)

Table 8-23 gives the significance of the I_DVS bits that govern the connect option already described.

Table 8-23. I_DVS Word in Connect IORB (ROP Mode)

Bit Number	Bit Value	Meaning for Connect Function
2	0	Do not use auto dial
	1	Use auto dial
10	0	Do not select ROP mode
	1	Select ROP mode

Disconnect Function

An application uses the disconnect IORB to terminate ROP mode processing.

The following paragraphs describe the options that an application can specify with a disconnect order.

ABORT QUEUED ORDERS

If the abort option is specified, outstanding IORBs (active and queued) are terminated with a device unavailable status (010B). The disconnect order is immediately serviced. If the abort option is not specified, all outstanding IORBs are allowed to complete before the disconnect order is serviced.

HANG UP

If the hang-up option is selected, the terminal is physically disconnected when the disconnect order is serviced. If the hang-up option is not specified, the communications connection remains active after servicing of the disconnect order (i.e., the terminal is logically disconnected, but remains physically connected).

Disconnect IORB (ROP Mode)

Table 8-24 gives the significance of the I_DVS bits that govern the disconnect options already described.

Table 8-24. I_DVS Word in Disconnect IORB (ROP Mode)

Bit Number	Bit Value	Meaning for Disconnect Function
14	0	Abort outstanding requests
	1	Wait until outstanding requests complete before disconnecting terminal
15	0	Hang up the phone
	1	Do not hang up the phone

Write Function (ROP Mode)

The write order is used to transmit data to the serial printer.

Once the LPH has verified the buffer range and address in the IORB, it performs control byte processing (if specified in I_DVS). ATD then services the write request. The data written can be of any length; using the ETX/ACK protocol, ATD sends the data a block-at-a-time to the printer.

CONTROL SEQUENCES

An application can control the write operation by means of control sequences imbedded in transmitted data.

DLE EOT Control Sequence

Write orders to a PRU7070 or 7075 support a feature that is useful to the application designer. If DEL (10) EOT (04) are the last two characters in the output data buffer, the write order that refers to this buffer is posted back to the application only when the device has printed the entire contents of the buffer. If DEL EOT are not supplied at the end of the buffer, the write order is posted back when the printer (by means of an ACK response) declares itself ready to receive the last block of buffer data. Conceivably, the device could fail to print the last block after receiving it. Thus, the DEL EOT sequence provides assurance that the entire buffer is actually printed.

Other Sequences

An application can place in the data buffer the customary serial printer control characters (e.g., carriage return, line feed, horizontal tab).

Other serial printer command and control sequences are available to the application. These can be used to change such printing characteristics as type pitch (number of characters per inch) and number of lines per inch. The user should consult the appropriate device manual for a more detailed discussion of printer control sequences.

Prohibited Sequences

An application cannot place in the data buffer the ETX (X'03') or ENQ (X'05') control characters, which are used by the ETX/ACK protocol. Nor, when transmitting data to a PRU7070 or 7075 printer, can the application send the following escape sequences: RIS (1B63), KBL (1B5B58), or KBU (1B5B57). In either case, the LPH suppresses transmission of these sequences in order to maintain the integrity of the ETX/ACK transmission procedure.

WRITE OPTIONS

An application selects the following options by setting bits in the device-specific word (I_DVS) of the write order.

Control Byte

Through the use of a control byte, an application can specify the customary pre-order control operations. If present, the control byte is the first byte in the output buffer. The application indicates its presence by setting a bit in I_DVS. The application must also include the byte in the range (I_RNG) of the data to be transmitted. For a detailed description of the control byte option, including control byte format, see "Control Byte Processing" earlier in this section.

Line Feed and Carriage Return

Specifying in I_DVS the line feed and/or carriage return option causes, respectively, a line feed and/or carriage return to be sent to the printer when the write order completes.

Write IORB (ROP Mode)

Table 8-25 gives the significance of bits of the write I_DVS word that are applicable to a ROP mode write order.

Table 8-25. ATD Word I_DVS in ROP Mode Write IORB

Bit Number	Bit Value	Meaning for ROP Write Function
4	0	Include control byte
	1	Do not include control byte
11	0	Do not send post-order line feed
	1	Send post-order line feed
12	0	Send post-order carriage return
	1	Do not send post-order carriage return

Read Function (ROP Mode)

The read order is used to obtain status information from the serial printer. Two types of read orders can be issued: normal status read and attention status read. An application indicates in I_DVS the type of read desired.

NORMAL STATUS READ

When an application issues a normal status read order, the IORB field I_ADR must point to a 10-byte buffer. Upon completion of the read order, this buffer contains a device identifier and may additionally contain status information.

PRU1004 and 7007 printers provide a device ID in the first byte of the status buffer; the remaining bytes are unused.

PRU7070 and 7075 printers provide device status information in addition to the device ID, which is supplied in the first status byte. Refer to the appropriate serial printer manual for additional information on device status.

Table 8-26 summarizes the device IDs that are returned in response to a status read request.

Table 8-26. Device IDs for Serial Printers

Printer	Device ID
PRU1004	21
PRU7007	22
PRU7070	31
PRU7075	32

ATTENTION READ

This option applies only to the PRU1004 and 7007 printers; it informs the application when a device has gone off-line or has been reset by the operator. The status buffer is not updated to reflect the device ID of the printer.

If this option is specified, the read order is returned to the issuing application only when:

- The printer runs out of ribbon
- The printer runs out of paper
- The printer's reset switch is pressed.

Read IORB (ROP Mode)

Table 8-27 shows the signficance of bits of the I_DVS word that are applicable to a ROP mode read order.

Table 8-27. ATD Word I_DVS in ROP Mode Read IORB

Bit Number	Bit Value	Meaning for ROP Mode Read Function
0	0	Normal status read
	1	Attention status read (PRU1004 and 7007 only)

Status Codes Returned in I_CTl (ROP Mode)

ATD returns status codes in I_CTl and I_ST. The status code returned in I_CTl consists of the right byte of the status returned in the Rl register (when the I/O order completes). A status code often has more than one possible meaning. As explained later in "Status Information under I_ST", a user can determine a specific meaning by referring to word I_ST. For example, the status 0104, in itself, can mean illegal printer command, zero buffer address, or zero buffer range. If bit 13 of I_ST is set to 1, the status 0104 means zero buffer address. If, however, bit 14 of I_ST is set to 1, the status 0104 means zero buffer range.

SUCCESSFUL COMPLETION (0000)

A zero status (in I_CTl) indicates successful completion of the order. A write order IORB can additionally indicate (in I_ST) a device attention condition (initiated by the operator) on PRU7070 and 7075 printers. This condition in no way interferes with successful completion of this or subsequent orders placed against the printer. By initiating a device attention condition, the operator can directly interact with the application that is controlling the serial printer.

INVALID ARGUMENT STATUS (0104)

This status is returned for the following reasons:

- In write IORB
  - Illegal printer command (PRU7070, 7075). Consult appropriate printer manual for details.
  - Zero buffer address.
  - Zero buffer range.
- In read IORB
  - Read buffer less than 10 bytes long.

DEVICE NOT READY STATUS (0105)

An order is posted back with this status when a PRU7070 or 7075 printer is in an off-line state. The LPH issues this error status once; subsequent or outstanding write orders are serviced when the device is put in an on-line, operational state. The reported off-line condition is usually caused by the printer running out of ribbon or paper.

HARDWARE ERROR STATUS (0107)

This status is returned in a write order for the following reasons:

- Hardware printer fault (PRU7070, 7075)
- Failure of printer to respond to print or status commands (PRU7070, 7075).

Status Information in I_ST

The bit settings in I_ST qualify the status codes returned in I_CT1, as shown in Table 8-28. The first column of the table gives the bit in I_ST; the second column gives the status code returned in I_CT1; the third column shows the significance of the status code (column 2) when the I_ST bit (column 1) is set to one.

Table 8-28. IORB Word I_ST (ROP Mode)

Bit	Return Status	Meaning When Bit Set to 1
3	0000	Operator initiated attention (PRU7070, 7075)
	0104	Illegal printer command (PRU7070, 7075)
	0105	Device off-line (PRU7070, 7075)
	0107	Hardware printer fault (PRU7070, 7075)
13	0104	Zero buffer address
	0105	Paper out (PRU7070, 7075)
	0107	No response by device to print or status commands (PRU7070, 7075)
14	0104	Zero buffer range
	0105	Ribbon out (PRU7070, 7075)

Error Processing

The length of the write order determines how an application should react to a 0105 (device not ready) status when addressing a PRU7070 or 7075 printer. If the data to be written is less than the device block or frame size (typically 125 or 253 characters, depending on the printer model), the application can assume that the data will be completely printed when the device goes to an on-line state. Otherwise, the application should not assume that the data will be completely printed.

The application should assume that the device will fail to complete any print operation when a 0107 (hardware error) status is reported.

Timeout Processing

There are no timeouts for read or write orders.

STREAM MODE

Stream mode is used mainly for the transfer of files:

- To an application from a paper tape reader
- From an application to a paper tape punch
- Between cooperating applications.

Stream mode requires, minimally, a half-duplex communications line. If full X-ON/X-OFF flow control (described later under "Flow Control") is desired, a full-duplex communictions line must be used. This mode supports data transfer at rates of up to 9600 bits per second.

Four I/O request blocks (IORBs), of standard length, are supported: connect, disconnect, read, and write.

Connect Function

An application selects the stream mode of ATD by setting bit one of I_DVS to one.

The following stream mode options can also be specified with a connect order.

AUTO CALL

An application can specify the auto call option. Specifying auto call in I_DVS enables an application to establish a connection using either an 801-A or an 801-C ACU data set.

CONFIGURATION MASK

Bits six and seven of I_DVS in the connect IORB allow an application to choose between transmision of seven or eight bit data. The same bits also allow an application to choose between odd parity check, even parity check, or no parity check. Settings of the configuration mask override the type of parity check established by the ASD directive when the system was configured.

Connect IORB (Stream Mode)

Table 8-29 gives the significance of the I_DVS bits that govern the connect options already described.

Table 8-29. I_DVS Word in Connect IORB (Stream Mode)

Bit Number	Bit Value	Meaning for Connect Function
1	0	Do not use stream mode
	1	Use stream mode
2	0	Do not use auto dial
	1	Use auto dial
6,7	00	Seven bit data; no parity check
	01	Seven bit data; odd parity check
	10	Seven bit data; even parity check
	11	Eight bit data; no parity check

Disconnect Function (Stream Mode)

An application uses the disconnect IORB to terminate stream mode processing.

The following paragraphs describe the options that an application can specify with a disconnect order.

ABORT QUEUED ORDERS

If the abort option is specified, outstanding IORBs (active and queued) are terminated with a device unavailable status (010B). The disconnect order is immediately serviced. If the abort option is not specified, all outstanding IORBs are allowed to complete before the disconnect order is serviced.

HANG UP

If the hang-up option is selected, the terminal is physically disconnected when the disconnect order is serviced. If the hang-up option is not specified, the communications connection remains active after servicing of the disconnect order (i.e., the terminal is logically disconnected, but remains physically connected).

Disconnect IORB (Stream Mode)

Table 8-30 gives the significance of those I_DVS bits that govern the disconnect options already described.

Table 8-30. I_DVS Word in Disconnect IORB (Stream Mode)

Bit Number	Bit Value	Meaning for Disconnect Function
14	0	Abort outstanding requests
	1	Wait until outstanding requests complete before disconnecting terminal
15	0	Hang up the phone
	1	Do not hang up the phone

Read and Write Functionality

Stream mode input/output functions can best be understood as an interaction between a transmitter and receiver. For this reason, read and write functionalities are initially discussed together.

CONTROL BYTE (STREAM MODE)

Stream mode supports a different type of control byte processing than do the the other modes of ATD. In the other modes of ATD, the control byte controls pre-order processing; in stream mode, the control byte is used to:

1. Return the status of a read or write order to the issuing application.
2. Control the execution of read and write orders.

The second use of the control byte allows an application to directly control certain aspects of stream mode processing even if the application is accessing the LPH through the File System (rather than through Physical I/O). The second use of the control byte is described later in this section under "Read Function" and "Write Function". The following paragraphs concern first use of the control byte.

On completion of a read or write order, the stream control byte contains a value that indicates the return status of the completed order. Table 8-31 correlates control byte values with standard system codes returned in register R1 and I_CTl.

Table 8-31. Stream Control Byte Return Codes

Return Code	Stream Control Byte	Meaning
0000	00	Successful completion
0104	44	Invalid parameter
0106	46	Device timeout
0107	47	Hardware error
010B	4B	Device unavailable
010F	4F	End of file

PROCESSING OF CONTROL BYTE AND DEVICE SPECIFIC WORD

As mentioned above, the stream control byte can be used, like the IORB device specific word I_DVS, to specialize execution of an I/O order. The processing of control byte and I_DVS information varies between read and write orders.

When a set of read orders are issued to read an entire file, I_DVS and the stream control byte of only the first read order are used to control the read operation. The information supplied by the first read order is used to process subsequent read orders (regardless of the their I_DVS or stream control byte settings) until the entire file has been received.

This particular method of processing is necessary because, as explained later in the example of file transmission, stream mode must be able to accept incoming data after the first read in the absence of subsequent read orders. Because the incoming data must be handled, the information for processing the data must be taken from the control byte and I_DVS that were specified by the first read order.

When, however, a set of write orders is issued to write an entire file, each write order is processed according to the I_DVS word and control byte supplied with that order. The one-time use of I_DVS and control byte information, described above, does not apply.

FLOW CONTROL PROTOCOL

Stream mode supports an optional X-ON (DC1)/X-OFF (DC3) protocol to maintain the orderly transmission of data.

Protocol Operation

The receiver indicates readiness to receive data by issuing an X-ON to the transmitter. After transmission has begun, the receiver can at any time interrupt transmission by issuing an X-OFF to the transmitter. The transmitter can resume sending data after receiving an X-ON from the receiver. After transmitting all the data, the transmitter can send an X-OFF to signify the end of transmission.

In sum, the receiver can solicit data by issuing an X-ON and suspend the transmission of data by issuing an X-OFF. The receiver solicits transmission when a buffer is available to store incoming data and suspends transmission when a buffer is not available.

Protocol Combinations

By means of stream control byte and I_DVS values, four combinations of flow control can be established.

1. Transfer is solicited and suspendable.

2. Transfer is not solicited but is suspendable.

3. Transfer is solicited but is not suspendable.

4. Transfer is not solicited and is not suspendable.

If transfer is suspendable, a full-duplex communications line is required; otherwise, a half-duplex line can be used.

Suspendable transfer is recommended for configurations that operate at high line-speeds, where there is an increased possibility of data loss due to the receiver's inability to supply read orders in a timely fashion.

Table 8-32 lists recommended combinations of line control according to line speed and type of operation.

Table 8-32. Recommended Line Control Combinations.

Operation		Line Speed	Recommended Combination
Receiver	Transmitter		
Application	Application	High	1
Application	Paper Tape Reader (Auto Start)	High	1
Application	Paper Tape Reader (Manual Start)	High	2
Paper Tape Punch	Application	High	2
Application	Paper Tape Reader (Auto Start)	Low	3
Application	Paper Tape Reader (Manual Start)	Low	4
Paper Tape Punch	Application	Low	4

CONTROL CHARACTERS

The signficance of control characters DC1 and DC3, when issued by the receiver, has already been discussed under "Flow Control". The following control characters are issued by the transmitter.

Control Charcter (Hex)	Meaning
Carriage return (0D)	End of Record
DC3 (13)	End of file
Backslash (5C)	Hide
Del (7F)	Pad

When stream mode encounters a carriage return, indicating end of record, it posts the current read IORB as successful (zero return status) with a non-zero residual range status in I_ST1.

Stream mode interprets DC3 as indicating both end of record and end of file. It posts the current read order back to the receiving application with nonzero residual range status in I_ST1. It posts the next read IORB with an end of file status (0F in I_CT1).

Assuming that the edit option has been selected, stream mode intreprets the backslash, or hide control character, as a signal to treat the next character as data, without interpretation. For example, if a backslash precedes a carriage return, ATD treats the carriage return as data instead of as an end of record indicator. (The edit option is explained in the next subsection.)

The LPH discards a DEL encountered in the input unless it is preceded by a hide character. The assumption is that an unidden DEL is either a rubout from a paper tape device or a pad character from an application.

EDIT OPTION

If the edit option is selected by both the transmitter and receiver, stream mode performs the following:

- On the transmit side, precedes with a backslash all non-printable ASCII characters (X'00' to X'1F' and X'7F').
- On the receive side, discards any transmitted hide character and accepts the following character as data. All unhidden non-printable ASCII characters are discarded.

To establish the edit option, the read application must select it in the first read order, by means of I_DVS or the control byte. The write application must select the option, by the same means, with each write order.

If the edit option is not selected on the transmit side, the LPH will not edit with the backslash all nonprintable ASCII characters. If the edit option is not selected on the receive side, the LPH will accept most characters (as either data or stream control characters). Only the DEL character is unconditionally removed from the incoming data stream.

FILE TRANSFER

The principal use of stream mode is to transfer files. A file, in stream mode, is a collection of records, each terminated by a (non-hidden) carriage return. The end of file is indicated by a DC3 character.

The following example of file transfer between two applications (called A and B) illustrates the stream mode functions thus far described. The example is illustrated by Figure 8-2.

It is assumed that both applications have issued connect orders specifying stream mode.

From Figure 8-2, it can be seen that the first read order issued by A specifies how the file is to be read. A control byte is to be used; transfer operations must be solicited and can be suspended. Application B specifies the same processing options, but does so with each write order.

Because the transmitting side indicates support for solicited transmission, the first write order does not initiate transmission of data until an X-ON (DC1) is received. The X-ON signifies that the receiver is ready to accept data.

The carriage return and line feed in the first write buffer are recieved as data (rather than control characters) because the control bytes set up by both applications specify edited transmissions.

ATD issues a carriage return when all the data in the first buffer has been transmitted, causing the first read order to be posted back to application A. The carriage return was specified in the control byte of the first write order.

Because the LPH does not immediately receive a second read order, it issues an X-OFF to suspend data transmission. However, between the posting of the first read order and the issuance of DC3, the transmitting side has sent the first three characters of the second write order (JKL). These characters are stored in the receive-side Multi-Line Communications Processor (MLCP) and edited according to the control byte and I_DVS supplied with the first read order.

After a second read order has been issued, ATD sends an X-ON, causing the resumption of data transmission.

The two DELs in the second write buffer are transmitted because the second write control byte (unlike the first) specifies non-edited transmissions. On the receive side, however, stream mode strips out the DELs because they are not preceded by the hide character.

There is again a delay in the transmission of data. When the third write order is ultimately issued, an X-ON is sent by ATD on the receive side to initiate transfer of data from the buffer associated with the third write order.

The stream control byte associated with the third write buffer directs ATD to send an end of file (DC3), which in turn causes the receive side ATD to post back the third read buffer with data received before the DC3. The read application ultimately detects the end of file condition after the fourth read is issued. This read is posted back with an 010F status and a stream control byte of 47, both of which indicate an end of file condition.

```
          RECEIVE SIDE                                    TRANSMIT SIDE

APPLICATION          ATD STREAM                    ATD STREAM          APPLICATION
 PROGRAM A             MODE                         MODE                PROGRAM B

Issue Read #1 ->
 DSW = C000
 BUF = @...
 SCB = 40                                                          <- Issue Write #1
                                                                       DSW = 00C0
                                                                       BUF = @ABCcr1f
                                                                       SCB = 40
                     *Send dc1      ->
                                       <-   Send A
                                       <-   Send B
                                       <-   Send C
                                       <-   Send \                      :
                                       <-   Send cr                     :
                                       <-   Send \                      :
                                       <-   Send lf
                                       <-   Send cr
                <-  Post Read #1                          Post Write #1 ->
                     STS = 0000                            STS = 0000
                     BUF = @ABCcr1f                        BUF = @ABCcr1f
                     SCB = 40                              SCB = 40
                                                                    <- Issue Write #2
                                       <-   Send J                     DSW = 00C0
                                       <-   Send K                     BUF = @JKLMNdeldel
                    **Send dc3      -> <-   Send L                     SCB = 42
                          .                   .
                          .                   .
                          .                   .
Issue Read #2 -> **Send dc1      ->                                     :
 DSW = C000                            <-   Send M                     .
 BUF = ....                            <-   Send N
 SCB = n/a                             <-   Send del
                                       <-   Send del
                                       <-   Send cr
                <-  Post Read #2                          Post Write #2 ->
                     STS = 0000                            STS = 0000
                     BUF = @JKLMN                          BUF = @JKLMNdeldel
                     SCB = 40                              SCB = 40
                                                                    <- Issue Write #3
                                       <-   Send V                     DSW = 00C0
                                       <-   Send W                     BUF = @VWXYZ
                    **Send dc3      -> <-   Send X                     SCB = 41
                          .                   .
                          .                   .
                          .                   .
Issue Read #3 -> **Send dc1      ->
 DSW = C000                            <-   Send Y
 BUF = ....                            <-   Send Z
 SCB = n/a                             <-   Send dc3
                <-  Post Read #3                          Post Write #3 ->
                     STS = 0000                            STS = 0000
                     BUF = @VWXYZ                          BUF = @VWXYZ
                     SCB = 40                              SCB = 40
Issue Read #4 ->                                                    :
 DSW = C000     <-  Post Read #4                                    :
 BUF = ....         STS = 010F
 SCB = n/a          BUF = ....
                    SCB = 47
```

NOTES

```
DSW = Device Specific Word          cr  = Carriage Return
SCB = Stream Control Byte           lf  = Line Feed
STS = Status                        n/a = Not Applicable
BUF = Buffer                        *   = Only if Solicited
\   = Hide Control Character              Option Selected
dc1 = DC-1 (X-ON)                   **  = Only if Suspendable
dc3 = DC-3 (X-OFF)                        Option Selected
del = Delete
```

Figure 8-2. Sample File Transfer Operation

Read Function

The read order is used to receive data from a paper tape device or cooperating stream mode application. Read order options are described in the following paragraphs.

SOLICITED TRANSFER

If this option is selected and a read buffer is available, stream mode sends an X-ON (DC1) to the transmit side, indicating that the receiving side can accept data.

SUSPENDABLE TRANSFER

If this option is selected, and there is no read buffer available to store the incoming data, and at least three transmitted characters have been received, stream mode sends an X-OFF (DC3) to the transmit side. When a read buffer becomes available, stream mode issues an X-ON (DC1).

CONTROL BYTE

The echo and edit options can be selected by setting bits of the control byte, as shown in Table 8-33.

Table 8-33. Read Order Stream Control Byte

Bit Number	Bit Value	Meaning
3	0 1	Do not echo received data Echo received data
6	0 1	Edit received data Do not edit received data

If the control byte option is specified, the first byte of the read buffer is assumed to contain a stream control byte.

The setting of a stream control byte overrides the corresponding bit setting in I_DVS. The setting is significant only in the control byte associated with the first read IORB; subsequent settings are ignored.

The user can take advantage of the fact that control byte bits 0 through 2 are not defined when specifying control byte settings. By setting bit 1 to one and bits 0 and 2 to zero, the user can specify a control byte that is in the printable ASCII range. For example, a control byte with bit 1 set to one and specifying no edit (bit 6 set to one) is equivalent to 42 (an ASCII letter B).

When the read operation is complete, the stream control byte contains the status of the completed order (refer to "Stream Control Byte Support" earlier in this section).

ECHO

If this option is specified, all received data is echoed or "reflected" back to the transmitter. If the option is not specified, received data is not echoed back to the transmitter.

EDIT

If this option is specified, a hide control character in the data stream is discarded and the succeeding character is treated as data. The latter can be any character from the ASCII character set, including the DEL character. If the nonprintable ASCII characters X'00' to X'0C', X'0E' to X'1F', and X'7F' are not preceded by the hide control character, they are discarded from the data stream. The nonprintable character X'0D' (carriage return) is treated as an end of record indicator when not preceded by a hide control character.

If the option is not specified, the hide control character is treated as a data character having no special meaning, and only DEL characters will be removed from the data stream.

Read IORB (Stream Mode)

Table 8-34 gives the significance of I_DVS bits that govern the read options already described.

Table 8-34. I_DVS Word in Read IORB (Stream Mode)

Bit Number	Bit Value	Meaning for Connect Function
0	0 1	Transfer is not solicited Transfer is solicited
1	0 1	Transfer is not suspendable Transfer is suspendable
2	0 1	Stream control byte is supported Stream control byte is not supported
3	0 1	Do not echo received data Echo received data
6	0 1	Edit received data Do not edit received data

Write Function

The write order is used to transmit data to a paper tape device or cooperating steam mode application. Write mode options are described in the following paragraphs.

SOLICITED TRANSFER

If this option is specified, stream mode does not begin transmission of the first record of a file until an X-ON is received. If this option is not specified, stream mode transmits data without waiting for the X-ON signal.

SUSPENDABLE TRANSFER

If this option is specified, stream mode suspends transmission of data when an X-OFF is received. The LPH resumes transmission upon receipt of an X-ON.

If this option not specified, transmission proceeds without regard to the receiver's ability to accept data.

LINE FEED AND CARRIAGE RETURN

If the line feed or carriage return option is specified, a line feed or carriage return, respectively, is sent at the end of a write operation. The carriage return is a signal to the receiver; it is not accepted by the receiver as data.

EDIT

If this option is specified, all non-printable characters (X'00' to X'1F'), the hide character (X'5C'), and the DEL character (X'7F') are preceded by an inserted backslash. If the option is not specified, these characters are transmitted without a backslash. The action taken by the receiver depends upon receiver's choice of options in the first read order. For further details, see "Read Function" earlier in this section.

END OF FILE

If this option is selected, stream mode sends a DC3 (end of file) at the end of the current write order, indicating to the receiver that the file has been completely transmitted.

CONTROL BYTE

The write options just described can be selected by setting bits in a control byte, as shown in Table 8-35.

Table 8-35. Write Order Stream Control Byte

Bit Number	Bit Value	Meaning
3	0 1	Do not send line feed at end of write Send line feed at end of write
4	0 1	Send carriage return at end of write Do not send carriage return at end of write
6	0 1	Edit transmitted data Do not edit transmitted data
7	0 1	Do not send DC3 (end of file) at end of write Send DC3 (end of file) at end of write

If the control byte option is specified, the first byte of the write buffer is assumed to contain a stream control byte.

The setting of a stream control byte overrides the corresponding bit setting in I_DVS.

The user can take advantage of the fact that control byte bits 0 through 2 are not defined when specifying control byte settings. By setting bit 1 to one and bits 0 and 2 to zero, the user can specify a control byte that is in the printable ASCII range. For example, a control byte with bit 1 set to one and specifying no edit (bit 6 set to one) is equivalent to 42 (an ASCII letter B).

When the write operation is complete, the stream control byte contains the status of the completed order (see "Stream Control Byte Support" earlier in this section).

WRITE IORB (STREAM MODE)

Table 8-36 gives the significance of I_DVS bits that govern the write options already described.

Table 8-36. I_DVS Word in Write IORB (Stream Mode)

Bit Number	Bit Value	Meaning for Write Function
8	0 1	Transfer is not solicited Transfer is solicited
9	0 1	Transfer is not suspendable Transfer is suspendable

Table 8-36 (cont). I_DVS Word in Write IORB (Stream Mode)

Bit Number	Bit Value	Meaning
10	0	Stream control byte is supported
	1	Stream control byte is not supported
11	0	Do not send line feed at end of write
	1	Send line feed at end of write
12	0	Send carriage return at end of write
	1	Do not send carriage return at end of write
14	0	Edit transmitted data
	1	Do not edit transmitted data
15	0	Do not send DC3 (end of file) at end of write
	1	Send DC3 (end of file) at end of write

Stream Mode Configuration

If stream mode is to support suspendable data transfer, full-duplex communication facilities (communications lines and modems) must be used. If neither of these options are specified, half-duplex communications facilities can be used.

Stream mode supports data transfer at speeds ranging from 110 to 9600 bits per second.

Error Processing

When a parity or framing error is detected, an ASCII SUB character (X'1A') is stored in place of the received character that was in error. When processing terminates abnormally, the read order is posted with a hardware error (0107) status, and I_ST indicates the reason for the termination.

Timeout Processing

A timeout occurs on an active read order if any of the following conditions are not satisfied within 30 seconds:

- Read buffer is filled with received data
- Carriage return (end of record) is received
- DC3 (end of file) is received.

The timeout value for an active write order is also 30 seconds. This interval includes any time during which the transmitter is waiting for a DC1 or DC3 from the receiver.

Section 9
STD LINE PROTOCOL HANDLER

SYNCHRONOUS TERMINAL DRIVER (STD) LINE PROTOCOL HANDLER

The Synchronous Terminal Driver (STD) line protocol handler (LPH) supports synchronous polled terminals, and the asynchronous receive-only printers (ROPs).

The basic VIP consists of a cathode ray tube (CRT) display screen and keyboard, with a synchronous communications interface. Its operating speeds are as follows:

Device Type	Peripheral	Baud Rate
VIP7700	ROP	2000 to 4800
VIP7700R	ROP	2000 to 9600
VIP7804	ROP	2000 to 19200
VIP7760	ROP, DSK	4800 to 9600
VIP7740	ROP, DSK	9600
VIP7710	ROP	9600
TWU 1901		4800 to 9600

Receive-Only Printers

TN 1200
TN 300
PRU 1003
PRU 1005
PRU 1901

GENERAL STD LINE PROTOCOL HANDLER OPERATION

Software Functional Support for the VIP

The following STD line protocol handler software functions support the basic VIP terminal:

- Poll and select communications procedures
- Poll line control
  - Poll list
  - Poll interval
  - Poll list stall interval
- Multipoint configuration support
- Switched and private line operation
- Auto-answer for switched network operation
- Modem, direct connect, and modem bypass interconnection modes
- Message/block transfer to and from a CRT
- Master LRN processing
- Fully addressable CRT entry marker control
- Pre-editing (control byte) and post-editing (I_DVS)
- Transfer of hardware function code to and from the application
- Long Q frame
- Error recovery procedrues
- Break processing (VIP7804 only)
- Half-duplex line function
- 2/4 wire line function.

The following functions support added terminal options:

- User-controlled CRT forms mode
- Message/block transfer to receive-only printer (ROP)
- User-controlled storage and retrieval of forms on the diskette (7740 and 7760 only).

User-Supplied Software Functions for VIP Support

The application program must supply the following functions to support data exchange between the terminal and the application:

- User-specified device arguments (polling interval and, at system building, station addresses and device type).

For messages to the VIP terminal, the application should provide:

- Optional; hardware function codes (1, 2 for all VIP except 7804, which only uses 1)
- Complete message text, including all required format control characters
- Optional; pre-editing and post-editing characters within message text
- Mandatory; complete forms definition message text for forms mode.

For messages received from the VIP terminal, the application must provide:

- Interpretation of hardware function codes (1, 2 for all but VIP7804, which only uses 1)
- Message processing (complete message or block, with possible use of master LRN with either)
- Interpretation of format codes (LF, CR, HT, VT) in the message text.

STD Request Response Time

Table 9-1 shows how to calculate the request response times needed by the line protocol handler for the connect, read, and write functions for the listed devices.

USING THE STD LINE PROTOCOL HANDLER

STD-Specific IORB Values

The VIP-specific input/output request block (IORB) item I_CT2, device specific word I_DVS, and software status word I_ST are shown in Tables 9-2, 9-3, and 9-4, respectively. Bits not explicitly described in the tables must be 0. Section 4 describes the general form of the IORB.

Table 9-1. STD Line Protocol Handler Response Time

Function	Response Time	Device
Connect	5-minute timeout	Communications supervisor
Read/Write	The following equation is used to calculate the time required to send/receive a message to/from the CRT: M/C = T where: M = Message size (range) C = Number of characters per second (line speed) Possible values for C: 250 = 2000 baud 300 = 2400 baud 600 = 4800 baud 1200 = 9600 baud 2400 = 19.2 kilo baud T = Timeout value in seconds NOTE If M is less than or equal to C, then T = 2.	All devices
	The following equation is used to calculate the time required to send/receive a message to/from the ROP attached to a CRT: T + (M/R) = V where: T = CRT timeout value (from above) M = Message size (range) R = ROP transmit rate	ROPs attached to VIPs

Table 9-1 (cont). STD Line Protocol Handler Response Time

Function	Response Time	Device
Read/Write (cont)	Possible values for R: 10 = 100 baud 30 = 300 baud 120 = 1200 baud V = Total timeout value in seconds	

Table 9-2. Function Codes in I_CT2 of the IORB

Function Code	Definition	Use
1	Write	Used by the line protocol handler to complete the description of the requested I/O function.
2	Read	
A	Connect	
B	Disconnect	

Table 9-3. STD Device-Specific Word I_DVS in the IORB

Bit Number(s)	Bit Setting	Meaning of Bit Setting
For connect call only (function code A).		
0	0	No meaning.
	1	Terminal-generated block mode.
2	0	Do not use Auto Call Unit.
	1	Use Auto Call Unit

Table 9-3 (cont). STD Device-Specific Word I_DVS in the IORB

Bit Number(s)	Bit Setting	Meaning of Bit Setting
3	0	Set cursor to home position on page overflow (write request). (Not applicable to VIP7804.)
	1	Do not set cursor to home position on page overflow (write request). (Not applicable to VIP7804.)
4	0	Control word specified (read/write request).
	1	No control word specified (read/write request).
5, 6, 7		Logical poll interval (read request, polled lines only):
	000	Poll continuously.
	001	1-second poll interval.
	010	2-second poll interval.
	011	3-second poll interval.
	100	4-second poll interval.
	101	5-second poll interval.
	110	15-second poll interval.
	111	30-second poll interval.
8	0	No space suppress. (VIP7804 only.)
	1	Space suppress. (VIP7804 only.)
9	0	Roll. (VIP7804 only.)
	1	No roll. (VIP7804 only.)

Table 9-3 (cont). STD Device-Specific Word I_DVS in the IORB

Bit Number(s)	Bit Setting	Meaning of Bit Setting
A		For ROP attached to VIPs 7700, 7700R, and 7760.
	0	150/PRT ROP address.
	1	150/NUL ROP address.
B	0	Hardware function codes are not specified (write request).
	1	Hardware function codes are specified (write request). (Not allowable for VIP7804.)
C	0	Do no timeout, use logical poll interval (read request).
	1	Timeout immediately (read request).
D	0	Return key equals transmit (VIP7804 only).
	1	Return key equals normal (VIP7804 only).
For write call only (function code 1)		
0	0	No meaning.
	1	Abort write IORB subfunction.
3	0	No preemptive write.
	1	Preemptive write.
4	0	Include control byte.
	1	Do not include control byte.
5	0	Reserved for system use (must be zero).
6		Reserved for system use.
8	0	No meaning.
	1	If bit 9 = one, supervisory write with reset; otherwise, no meaning.

Table 9-3 (cont). STD Device-Specific Word I_DVS in the IORB

Bit Number(s)	Bit Setting	Meaning of Bit Setting
9	0	Normal message.
	1	Supervisory message, if no roll on connect.
A	0	RFU.
	1	RFU.
B	0	No line feed at end of message.
	1	Line feed at end of message.
C	0	Carriage return at end of message.
	1	No carriage return at end of message.
D, E, F		Number of copies to be printed (VIP 7804 only).
	000	1 copy.
	001	2 copies.
	010	3 copies.
	011	4 copies.
	100	5 copies.
	101	6 copies.
	110	7 copies.
	111	8 copies.
For read call only (function code 2)		
0	0	No meaning.
	1	Abort read IORB subfunction.
2	0	No meaning.
	1	ESC B Diskette request.

Table 9-3 (cont). STD Device-Specific Word I_DVS in the IORB

Bit Number(s)	Bit Setting	Meaning of Bit Setting
		For Read Call Only (cont)
9	0	Normal message.
	1	Supervisory message, if no roll on connect.
For disconnect call only (function code B)		
1	0	No meaning.
	1	Send DLE EOT (VIP 7804 only).
E	0	Purge outstanding requests and disconnect immediately.
	1	Wait until all requests are complete before disconnecting.
F	0	Hang up phone after disconnect.
	1	Maintain phone connection after disconnect.

Table 9-4. STD Software Status Word I_ST in the IORB

Bit	Contents of $R1	Meaning When Bit is Set to 1
1	0	ETB received
2	0	Data service error (transmit)
6	0	Long record received (receive)
7	0	Illegal character (transmit)
8	0	Sequence error (receive)
E	104	Range error
7	106	Read timeout
8	107	NAK limit reached
D	107	Page overflow
F	107	Busy
0	10B	Abort
2	10B	Data service error (receive)

Table 9-4 (cont). STD Software Status Word I_ST in the IORB

Bit	Contents of $R1	Meaning When Bit is Set to 1
7	10B	Illegal character (receive)
8	10B	Poll failure/sequence error
9	10B	Excessive checksum/parity errors (receive)
B	10B	Phone hang up
3	10B	Read timeout
E	10B	Not available

STD Polling Options

Polling (the line protocol handler's request to the VIP terminal on a polled line for data) is subject to four kinds of control: two specified at system build, and two specified at connect time. The former consists of the poll lists and poll list stall, while the latter are the poll interval and poll duration.

The application, at connect time, is required to specify the arguments for the poll interval and poll duration, by setting the appropriate bits in the IORB's device-specific word I_DVS (Table 9-3).

STD POLL LIST

The poll list specifies the station addresses to be used in the polling sequence. Multiple occurrences of a particular address may be used to increase the polling frequency of that address. The list is defined at system build (see the System Building and Administration manual).

STD POLL LIST STALL

Poll list stall is the delay interval, in seconds, between poll list cycles. This delay is specified at system build (see the Building and Administration manual.

STD POLL INTERVAL

The poll interval specifies the minimum period of time between each successive request (poll) by the line protocol handler for data from a VIP terminal. The line protocol handler will poll the VIP once for each read request, and when the request is not satisfied, again after the specified poll period elapses.

For example, with a 1-second poll interval, the line protocol handler will issue the same read request every second. For a zero poll interval, the line protocol handler will poll the VIP terminal continuously.

The application specifies the poll interval according to the bit settings of bits 5, 6, and 7 in the device-specific word I_DVS of the IORB, as shown in Table 9-3.

STD POLL DURATION (TIMEOUT)

Poll duration, or the timeout interval, is the maximum time that the line protocol handler will wait for polled data from the VIP, before discontinuing the read attempt and read request. The possible timeout intervals are immediate (i.e., after only one poll) and indefinite (i.e., until requested data is received). The application specifies the poll duration or timeout interval with the bits 5, 6, 7, and C in the connect device-specific word I_DVS, according to the bit values shown in Table 9-3.

STD LINE PROTOCOL HANDLER POLL FUNCTIONS

Within the parameters specified in the poll argument values by the application, the line protocol handler provides all necessary polling functions (e.g., how terminals share a common line, or which terminal is processed next based on the poll list).

When the application bypasses these line protocol handler poll functions (i.e., by specifying immediate timeout after only one poll), the application must then provide for proper operation and coordination among all terminals on the line.

When the application is to issue to the terminal (VIP 7804/7805) writes containing TXA or TXD escape sequences, the user should first issue an asynchronous read. The use of immediate timeouts on reads in this case could cause the read to be issued and posted before the write is queued and issued, resulting in a loss of data from the TXA or TXD command.

Polling is defined as the actual read, not the reading of the poll list. Polling itself does not commence unless a read has been queued. Only those stations on the poll list which have reads queued will be polled.

Control and Characterisitcs of STD Input (Keyboard/Screen)

STD INPUT MESSAGE HEADER

The line protocol handler strips the message header from the input data, except for the hardware function codes, and does not include the header in the application's buffer.

STD HARDWARE FUNCTION CODES

STD hardware function codes are listed in the appropriate hardware device manauals.

These codes provide a special message labeling capability to be used by the application. This capability does not apply to the VIP7804.

The application can include two function codes in the message header of each text message to or from a terminal by setting at connect time the following in the IORB: (1) set to 1, bit B of the device-specific word I_DVS (see Table 9-3); and (2) set to 1, bit B (extension bit) of I_CT2 to specify that the IORB is extended (see Figure 4-2 and Table 4-11). The line protocol handler then inserts the two user-specified hardware function codes at read time into the IORB's I_FCS word.

The VIP7804 has only one hardware function code that may be used by the application program. This function code appears as a two-character escape sequence in the data buffer. See the hardware manual.

STD INPUT DATA

The line protocol handler places into the application's buffer all data, between the STX and ETX/ETB control characters, received from the VIP terminal. The data is inserted into the buffer in 7-bit ASCII, with the most significant bit always zero. The LPH strips the ETX/ETB and LRC (longitudinal redundancy check character, see "Line Protocol Handler Functions," earlier) from the data and does not include them in the buffer.

Control and Characteristics of STD Output

This subsection pertains to VIP output and is applicable to the keyboard, display screen, or receive-only printer (ROP) as indicated.

STD OUTPUT MESSAGE HEADER

The STD line protocol handler supplies the output message header, but not the hardware function codes. Those for all but the VIP7804 may be supplied by the application as described above under "STD Hardware Function Codes."

At write time, when the hardware codes are specified, they are placed in the I_FCS word of the IORB. To write function codes to the VIP7700 hardware, the application program must, at connect time, set bit B (extension bit) of the IORB's I_CT2 word to 1, to specify that the IORB is extended. When they are not specified (i.e., bit 8 of I_DVS set to 0 at connect time), the line protocol handler will insert two spaces, instead of function codes 1 and 2, into the I_FCS word (see Figure 4-2 and Table 4-11).

CONTROL BYTE (SEND)

The control byte provides editing control (CR, LF, FF) for both ROPs and CRTs, as described later in this section.

STD OUTPUT DATA

The application's output data must be 7-bit ASCII (the eighth bit is ignored). Any ASCII control characters, if included in the application's data, are not transmitted.

STD KEYBOARD/SCREEN OUTPUT EDITING CONTROL

The line protocol handler sends LF and CR editing characters for VIP keyboard/screen devices according to the values of the B- and C-bits of the device-specific word I_DVS (Table 9-3). The application specifies these bit values at write time to send the CR and LF characters, as follows:

I_DVS Bits		Editing Characters
B	C	Sent
0	0	CR
0	1	None
1	0	LF, CR
1	1	LF

STD RECEIVE-ONLY PRINTER EDITING SEQUENCE

The line protocol handler sends an output editing character sequence for the receive-only printer (ROP) according to the control byte supplied, the values of the B- and C-bits of the device-specific word I_DVS (Table 9-3), and the VIP type to which it is attached. The application specifies these bit values at write time to send the ROP output editing sequence, according to the ROP type and the VIP type to which it is attached, as shown in Table 9-5.

STD RECEIVE-ONLY PRINTER CONTROL SEQUENCE

The STD line protocol handler sends an output control sequence according to the ROP type and the VIP type to which it is attached as shown in Table 9-6.

Table 9-5. STD Receive-Only Printer Editing Sequence

CRT Type	Attached ROP Type	I_DVS Bits B	I_DVS Bits C	Output Editing Sequence
All	All	0	0	CR
All	All	0	1	None
VIPs7700, 7700R 7760; VTS 7710, 7740	TN1200, PRU 1005	1	0	LF, CR, 36 DELs
VIPs7700, 7700R, 7760; VTS 7710, 7740	TN300, PRU 1003	1	0	LF, CR, 9 DELs
VIP7804	TN300, TN1200, PRU 1003, PRU 1005	1	0	LF, CR
VIPs7700, 7700R, 7760; VTS 7740 7710	TN1200, PRU 1005	1	1	LF, 36 DELs
VIPs7700, 7700R, 7760; VTS 7740, 7710	TN300, PRU 1003	1	1	LF, 9 DELs
VIP7804	TN300, TN1200, PRU 1003, PRU 1005	1	1	LF
TWU 1901		1	0	LF, CR

Table 9-6. STD Receive-Only Printer Control Sequence

CRT Type	Attached ROP Type	Form Feed	ZZZZ	Output Editing Sequence
VIP7804	All	X		FF
TWU 1901		X		FF
VIPs7700, 7700R, 7760; VTS 7740, 7710	TN300, PRU 1003	X		FF, 65 DELs

Table 9-6 (cont). STD Receive-Only Printer Control Sequence

CRT Type	Attached ROP Type	Form Feed	ZZZZ	Output Editing Sequence
VIPs7700, 7700R, 7760; VTS 7740, 7710	TN1200, PRU 1005	X		FF, 250 DELs
VIP7804	None	X		CLR
VIPs7700, 7700R, 7760; VTS 7740, 7710	None	X		FF, DEL
VIP7804	All		X	LF, CR
VIPs7700, 7700R, 7760; VTS 7740, 7710	TN300, PRU 1003		X	LF, CR, 9 DELs
VIPs7700, 7700R, 7760; VTS 7740 7710	TN1200, PRU 1005		X	LF, CR, 36 DELs
All	None		X	LF, CR

NOTES

1. Form feed (FF) is specified by bit 3 of the control byte.
2. ZZZZ represents the number of times an output editing sequence is performed and is specified by bits 4 through 7 of the control byte.
3. CLR clears all data attributes, moves the cursor to home position, and puts the terminal in text mode.

PRINTER ESCAPE SEQUENCE FOR VIP7804

The STD LPH transmits the following printer escape sequence before the first data message:

1B 5B 33 70 (PHLF)

and transmits the following printer escape sequence after the last data message:

1B 5B 3C 70 (PEOM)

Receive-Only Printer Support

Receive-only printer support by the STD LPH falls into three categories:

- VIP7804 attached ROP support
- VIP7700, 7700R, and 7760 attached ROP support
- PRU 1901 support.

For the VIP7804 attached ROPs, the STD inserts the start of message printer escape sequence (print host with local fill (PHLF)) before the first text message and appends the start print escape sequence (printer end of message (PEOM)) to the last text message. The application may supply the CR, LF, or FF characters minus the time fill characters in the text buffer, or may instruct the STD LPH to supply the CR, LF, or FF characters via the control byte or IORB device-specific word. Upon receipt of the CR, LF, or FF character, the VIP7804 printer adapter supplies the required time fill characters. For HT or VT, the application must supply the HT or VT character and required time fill characters in the text buffer. In this mode an extended print buffer of 132 print positions is available, as well as the option to have all text transparent. Use of any of the options provided by the VIP7804 printer adapter (e.g., copies option) requires the application to supply the appropriate escape sequence in the text buffer.

For the VIP7700, 7700R, and 7760 attached ROPs, the STD LPH supports the transparent (150 PRT) and nontransparent (150 NUL) print modes based on the setting of I_DVS bit A of the ROP connect IORB.

- Transparent mode: Allows the user to supply the CR, LF, or FF characters and timing fill characters in the text buffer, or instruct the STD to insert them. An extended print buffer of 132 print positions is also available in this mode.

- Nontransparent mode: The user need not include the CR or LF characters in the text buffer. The message received by the terminal is interpreted in the display format (80 print positions), and the necessary CR and LF characters are supplied by the terminal.

VIP7804 Support

While certain VIP7804 terminal operations are configurable by the application (e.g., roll, space suppress) via the connect IORB, the STD LPH imposes the following operational modes in order to ensure proper terminal operation:

- Block transmit auto: Successive blocks will be sent by the terminal, each time the terminal is polled, until the last block has been transmitted.

- Verify before process: The terminal normally operates in verify before process mode. In this mode, the terminal does not process the data unless the BCC indicates that no errors have occured. The transmitted data is restricted to 1024 characters, including all text plus the control characters CR, LF, FF, and DEL, supplied by both the application and the STD LPH.

- Process before verify: The user wishing to use blocks larger than 1024 characters must, at configuration time, specify PB after the 7804 device type. In this mode, the terminal displays characters as it receives them, without protecting the integrity of the screen.

TWU 1901 Support

The TWU 1901 is a synchronous, polled, hard-copy device with a keyboard. It should be configured as a CRT; the LPH will handle the addressing (150 PRT; 150 NUL).

Master LRN Processing

Master LRN processing enables one receive buffer to service up to a maximum of 32 terminals on a multi-dropped line. This technique drastically reduces the number of receive buffers required to support a multi-dropped environment. It is applicable only to read requests, and is supported through the user-supplied control word, (described below).

This feature is used most effectively when the application issues two or more asynchronous read requests, each specifying different buffers. The issuance of multiple read requests allows the application to process received data while the STD LPH polls another terminal for data. However, use of the master LRN feature does not guarantee that all terminals associated with the master LRN are accessed sequentially, since STD does not poll for data unless a read request has been queued. When data has been

received in this mode, STD returns the LRN, for which data was received, in the right half-byte (RHB) of the user-supplied control word. The application can then determine which terminal requires a response.

Sub-LRN Support

The ROP and diskette can be accessed only by sub-LRN. Access to ROPs is handled by the File System, assuming that the user entered the appropriate ROP and STDLN CLM directives at configuration time. To access the ROP at the physical I/O level, the application must set the sub-LRN in field I_ST to 1. If an error is returned and the same IORB re-issued, the sub-LRN must be reset, because STD might have returned a status in I_ST when posting the IORB, overwriting the original sub-LRN.

Block Mode Processing

Block mode processing is the transmission or reception of small data blocks, which are components of a large message. It conserves buffer space, conserves total message transmission time in the presence of errors, and reduces line errors. This mode, applicable to the VIP7760, CTS 7600, and VIP7804, is supported through the user-supplied control word.

In block mode transmit processing (ETB), a large message is transmitted in small blocks of data. The application is responsible for issuing an individual write request for each of these blocks. In block mode receive processing (ETB), the terminal, when polled, sends blocks of data until the last block is transmitted. The application, in this instance, is responsible for issuing the read request needed to initiate the transmission.

When I_DVS for the connect request specifies "terminal-generated block mode", the application must set the IORB range (RB_RA) to the size of the block expected. For the VIP7804, the RB_RA values are 20-2704 (i.e., 32-9999 decimal). For the VIPs7700, 7700R, and 7760, the RB_RA value is FF (256 decimal).

Control Word

Master LRN and block mode processing require an additional word at the beginning of the data buffer. The connect request specifies control word utilization, and the IORB buffer address (RB_ADR) contains the address of the control word.

If master LRN processing is desired, the right half-byte (RHB) of the control word must be set to the LRN which the user designated as the master. If master LRN processing is not desired, the application must set the RHB of the control word to zero (0). See Figure 9-1.

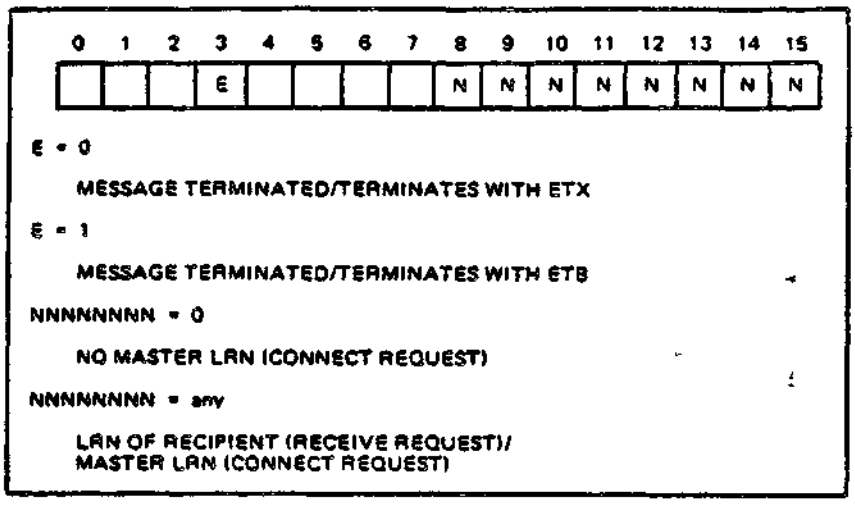


Figure 9-1. Control Word

Block mode processing requires that the application include the control word in the IORB range (RB_RA) of the read/write request. The IORB buffer address (RB_ADR) must contain the address of the control word. On write requests, if the data to be sent is a block (ETB), the application must set bit 3 of the control word. If the data to be sent is an entire message or the last block of a message, the application must set bit 3 of the control word to zero. On read requests, if the received message was terminated with ETB, the STD LPH sets bit 3 of the control word to 1. If the received message terminated with ETX, the STD sets bit 3 of the control word to zero. The use of the control word does not preclude the use of the control byte for printer editing. If the control word is used in conjunction with the control byte, the control word precedes the control byte in the buffer.

Control Byte

The control byte provides editing control (CR, LF, FF) for both ROPs and CRTs. This control is effected by setting bit 4 of the write IORB, which causes STD to treat the first character (third if the control word is specified) of the data as the control byte. The control type is examined by the STD and, according to the bit settings, the STD transmits the appropriate characters before the data. The control byte format is given in Figure 9-2.

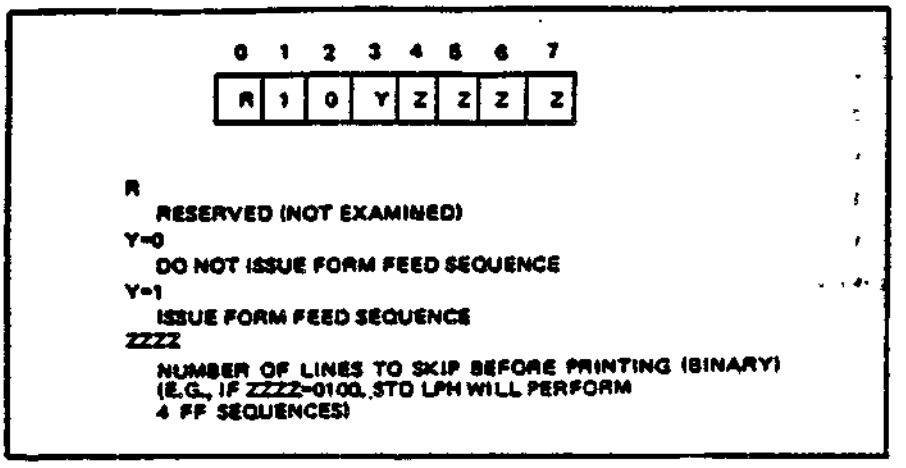


Figure 9-2. Control Byte

Output Data and Invalid Characters

The data must be 7-bit ASCII (the eighth bit is ignored). The ASCII control characters SOH (01), STX (02), and ETX (03), are not transmitted if included in the data. Instead, an SYN (16) is transmitted by the STD LPH in place of each occurrence.

VIP7804 Message Range Requirements (Verify Before Process Mode)

The maximum number of characters that the VIP7804 terminal (CRT/ROP) may have written to it is 1024 (verify before process). If the application specifies editing control for the CRT/ROP, the STD LPH inserts/appends the appropriate control characters (CR, LF, FF, PHLF, PEOM) to the data. While these characters do not appear in the data buffer and are not included in the IORB range, they do occupy terminal buffer space. Therefore, the application must account for these characters when issuing the data write request. Specifically, the data plus STD-generated transmitted characters must be less than or equal to 1024.

VIP7804 Terminal Transmission Modes and Cursor Positioning

The VIP7804 terminal supports two methods for transmitting data to the host and for positioning the cursor.

1. Return=Normal Mode: Data is transmitted from the terminal to the host by pressing the TRANSMIT key. The cursor is positioned to the next cursor position following the data to be transmitted. The RETURN key may be used to move the cursor to column 1 of the next line.

2. Return=Transmit Mode: Data is transmitted from the terminal to the host by pressing the RETURN key. Pressing the AUTO LINE FEED (AUTO LF) key changes the cursor's position after the data is transmitted, as follows:

   AUTO LINE FEED depressed.
   Cursor is positioned to column 1 of the next line.

   AUTO LINE FEED not depressed.
   Cursor is positioned to column 1 of the current line.

   The received range residue is modified to not reflect the reception of the CR/LF or CR.

VIP7804 Break Processing

Break processing on the VIP7804 is performed by the shifted function (F12) key rather than by the BREAK key used on other terminals.

When the terminal is operating in no-roll mode, pressing the shifted break function key causes the ** BREAK ** message to be displayed on the 25th line of the terminal. To respond to this message, the operator should:

1. Acknowledge the break message by pressing function key 10

2. Respond to the break condition by:

   a. Entering the UW, SR, or PI command, as appropriate

   b. Pressing the transmit key after entering the command.

Supervisory Messages

Supervisory message handling is applicable only to the VIP7804. To read or write supervisory messages, an application must first connect the terminal with bit 9 of I_DVS set to 1 (no roll).

SUPERVISORY MESSAGE READS

To read a supervisory message, the application must set bit 9 of I_DVS to one in the read IORB. Servicing of the supervisory read order places the cursor on the 25th line. To return the cursor from the supervisory message line to the data region of the screen, the operator must:

1. Press the return or transmit key (depending on the terminal's operating mode) in order to terminate the read.

2. Press function code 10.

SUPERVISORY MESSAGE WRITES

To write a supervisory messge, the application must set bit 9 of I_DVS to one in the write IORB. Servicing of the supervisory write order places the cursor and message on the 25th line.

If bit 8 of I_DVS in the write IORB is set to one, STD repositions the cursor to the location it occuppied before the supervisory write. If this bit is set to zero, the cursor remains on the 25th line until the operator presses function code 10.

Diskette Handling for the CTS 7760 and VTS 7740

The following conventions apply:

- The diskette cannot be accessed through the file system. The application must use physical I/O, setting I_ST to 2. The application must reset I_ST when re-using an IORB to issue an I/O order.
- Device specific words in connect, read, and write IORBs must indicate no control byte.
- The first two bytes of the application buffer must be one of the following escape sequences:

Escape Sequence	Meaning
1B 57	Write
1B 42	Read, display on terminal
1B 56	Read, send to host
1B 51	Erase

- To read or write buffers over 256 characters, an application must use ETB processing. Alternatively, CTS 7760 and VTS 7740 hardware allows the block size to be set at 128 characters.

2/4 Wire Line Function

Two types of wire connections are supported:

1. Two-wire: Two physical wires (one pair) make up the electrical circuit onto which a data set may be connected. There is a 250 millisecond data set turnaround time.
2. Four-wire: Four physical wires (two pairs) make up the electrical circuit onto which a data set may be connected. Four-wire does not infer full-duplex operation.

LONG Q FRAME LINE FUNCTION

ALL VIP terminal types supported by the STD LPH must be set to long Q frame (i.e., the Q-frame response by the terminal is SYN SYN SYN SYN SOH EOT).

ERROR PROCESSING BY STD LINE PROTOCOL HANDLER

Table 9-7 lists the errors reported by the STD line protocol handler for any VIP configuration. It also lists corresponding return status error codes (see Table 4-10), corresponding bits in the STD software status word I_ST (see Table 9-4), and possible recovery actions.

Error Condition	Posted Error Return Status	I_ST Bit	Possible Recovery	Comments
Error during open	B	As reported		
"Not available" message received	7	E	None	
Page overflow not corrected	7	D	None, or retry once	
Invalid range in IORB	4	E	None	
Read timeout	6	7	Immediate return	
NAK limit reached	7	8	Retry four times	
Busy received	7	F		
Purged due to immediate close or read/write abort	B	None		
Station disabled	B	None		
Data service rate error	0 (transmit) 7 (receive)	2 2, 8	Not applicable Retry four times	Not fatal
Long record	0	6	None (ACK sent to VIP)	Data lost
Illegal character	0 (transmit)	7	Replace illegal character with SYN characters	Bad character in application's buffer
Sequence error	B (receive)	8		
Phone hang up	B	B	None	
Excessive checksum or parity error	B	9	Retry four times	
Poll failure	B	8	Retry four times	

Section 10
PVE LINE PROTOCOL HANDLER

POLLED VIP EMULATOR (PVE) LINE PROTOCOL HANDLER

The PVE line protocol handler (LPH) allows a DPS 6/Level 6 system to be connected to a communications link that operates according to the polled VIP protocol. The line can be half or full duplex, dedicated, or switched, and operates at up to 9600 baud.

The PVE LPH also provides functionality to recognize and respond to the VIP7760 controller poll.

The computer that controls the communications link is known as the control station (CS), which can be any Honeywell host system that supports the VIP protocol.

GENERAL PVE LINE PROTOCOL HANDLER OPERATION

A PVE LPH, which is configured in a tributary processor, supports up to 32 tributary stations per line. Each tributary station appears to the control station as a VIP terminal.

To the control station, each PVE tributary station is known by a poll address, and to the tributary processor, by a logical resource number (LRN). There is a one-to-one relationship between the poll address and the LRN.

An application running in a tributary processor issues read and write requests against an LRN associated with a tributary station. Similarly, the control station communicates with a tributary station by issuing poll and selection orders with the appropriate poll or selection address.

Figure 10-1 illustrates a typical PVE configuration.

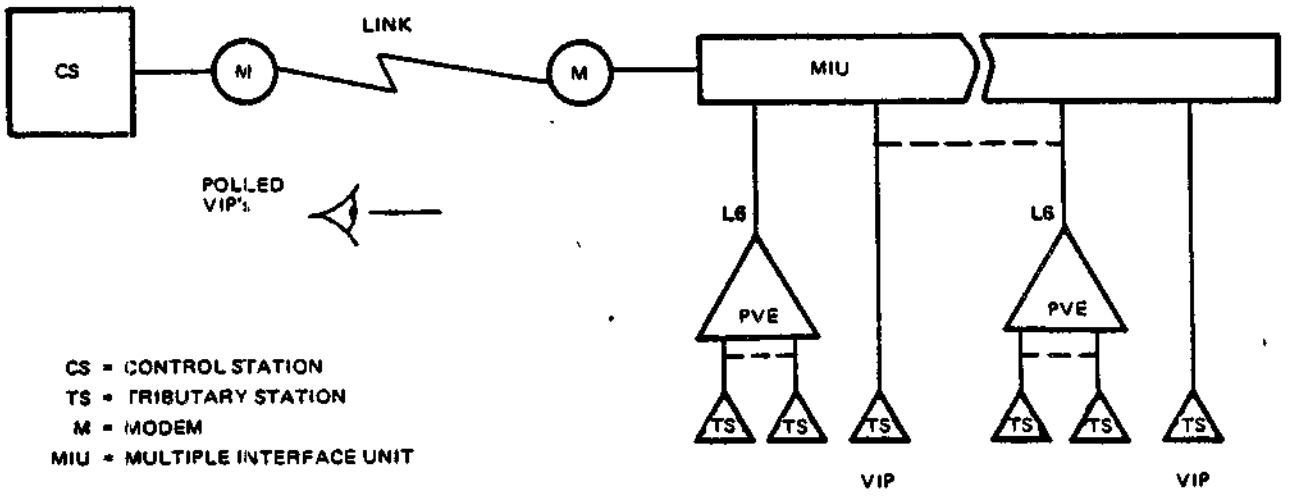


Figure 10-1. Typical PVE Configuration

When the PVE receives a select request with the LRN-associated poll address, it forwards the message to the tributary station to satisfy the application's read request. When the PVE receives a poll request for the LRN-associated poll address, it forwards the message to the control station to satisfy the application's write request. Thus, the application provides the equivalent of the screen and keyboard, with read and write requests, respectively.

The PVE line protocol handler supports only the screen and keyboard features of the VIP.

The PVE LPH also supports controller poll processing. This processing option, specified at system build, permits the PVE line protocol to support controller poll orders. Such orders are issued by the control station in support of a VIP7760 (CTS 7600) controller configuration. A typical controller configuration is shown in Figure 10-2. As many as eight controllers can be associated with a single communications link; up to 32 uniquely-identifiable stations can be associated with the controllers, grouped in any number under each controller. Each station so grouped, however, must have a unique poll address.

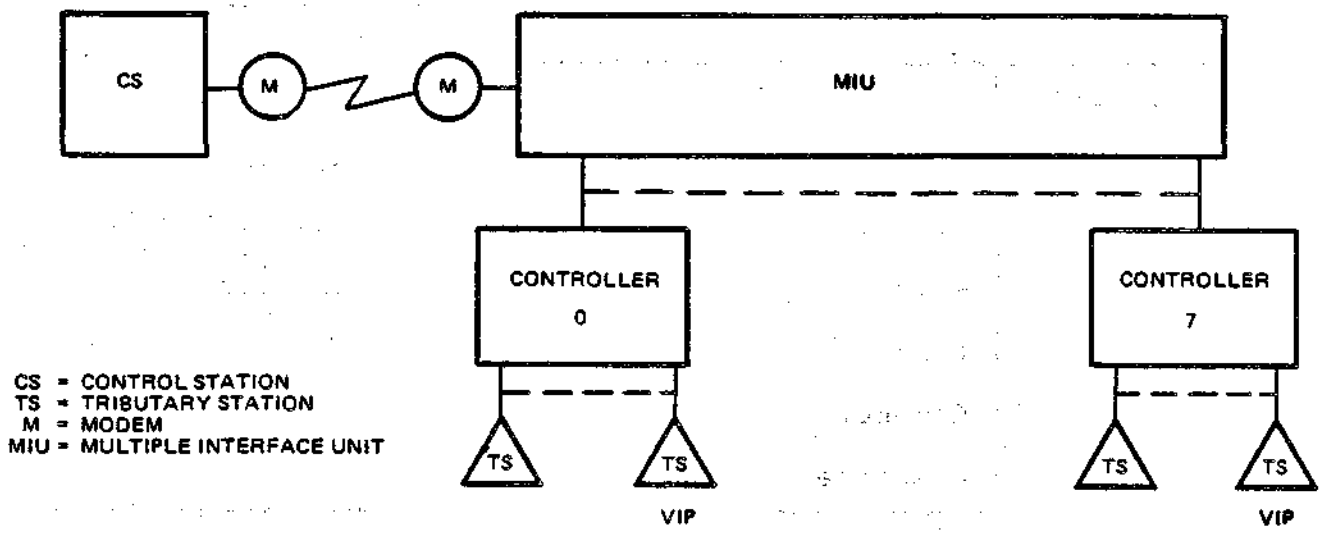


Figure 10-2. Typical Controller Poll Configuration

The advantage of controller poll processing is that the control station can issue a single controller poll message to a set of stations that are attached to the controller, instead of issuing individual sequential poll messages to the same set of stations. When the PVE receives a controller poll message, it individually checks all PVE stations that are associated with that controller. If any station has a write request pending, PVE forwards the message to the control station in response to the controller poll request.

USING THE PVE LINE PROTOCOL HANDLER

PVE-Specific IORB Values

The PVE-specific IORB item I_CT2, device-specific word I_DVS, and software status word I_ST are shown in Tables 10-1, 10-2, and 10-3, respectively. Bits not explicitly described in the tables must be 0. Section 4 describes the general form of the IORB.

Table 10-1. Function Codes in I_CT2 in the IORB

Function Code	Definition	Use
0	Wait online	Used by the line protocol handler to complete the description of the requested I/O function
1	Write	
2	Read	
A	Connect	
B	Disconnect	

Table 10-2. PVE Device-Specific Word I_DVS in the IORB

Bit Number	Bit Setting	Meaning of Bit Setting
0	0	Must be zero.
1	0	Must be zero.
For connect call only (function code A)		
2	0	Do not use Auto Call Unit.
	1	Use Auto Call Unit.
3	0	Must be zero.
4	0	Must be zero.
5	0	Must be zero.
6	0	Must be zero.
7	0	Must be zero.
8	0	Does not support VIP function codes.
	1	Supports VIP function codes.

Table 10-2 (cont). PVE Device-Specific Word I_DVS in the IORB

Bit Number	Bit Setting	Meaning of Bit Setting
8 (cont)		NOTE Inclusion of function codes in I_FCS by the application requires that bit B (extended IORB indicator) in I_CT2 be set to 1, specifying that the IORB is extended.
9	0	Must be zero.
A	0	Include received DEL characters in buffer.
	1	Strip received DEL characters.
B	0	Must be zero.
C	0	Must be zero.
D	0	Must be zero.
E, F		LPH response to application when LPH receives data but no read IORB available.
	00	Send NAK. (VIP Status Codes)
	01	Send ACK. (VIP Status Codes)
	10	Send BSY status. (VIP Status Codes)
	11	Send NAK (same as 00). (VIP Status Codes)
For disconnect call only (function code B)		
E	0	Abort (dequeue) all IORBs on request queue.
	1	Process all outstanding requests on request queue.
F	0	Hang up phone after disconnect.
	1	Do not hang up phone after disconnect.

Table 10-3. PVE Software Status Word I_ST in the IORB

Bit	Meaning When Bit Set to 1
0	N/A
1	N/A
2	Data service rate error
3	N/A
4	Communications control block (CCB) service error
5	N/A
6	Long record
7	0 = ETX character received 1 = ETB character received
8	NAK limit reached
9	Excessive checksum/parity errors
A	Nonzero residual range
B	Phone hang-up
C	N/A
D	N/A
E	N/A
F	Fatal error: bus parity or memory error

VIP Protocol Message Structure for PVE

Figure 10-3 shows two VIP protocol message structures for PVE.

Control and Characteristics of PVE Input

PVE INPUT MESSAGE HEADER

The PVE line protocol handler strips the message header, between the SOH and STX control characters, and does not include it in the application's buffer.

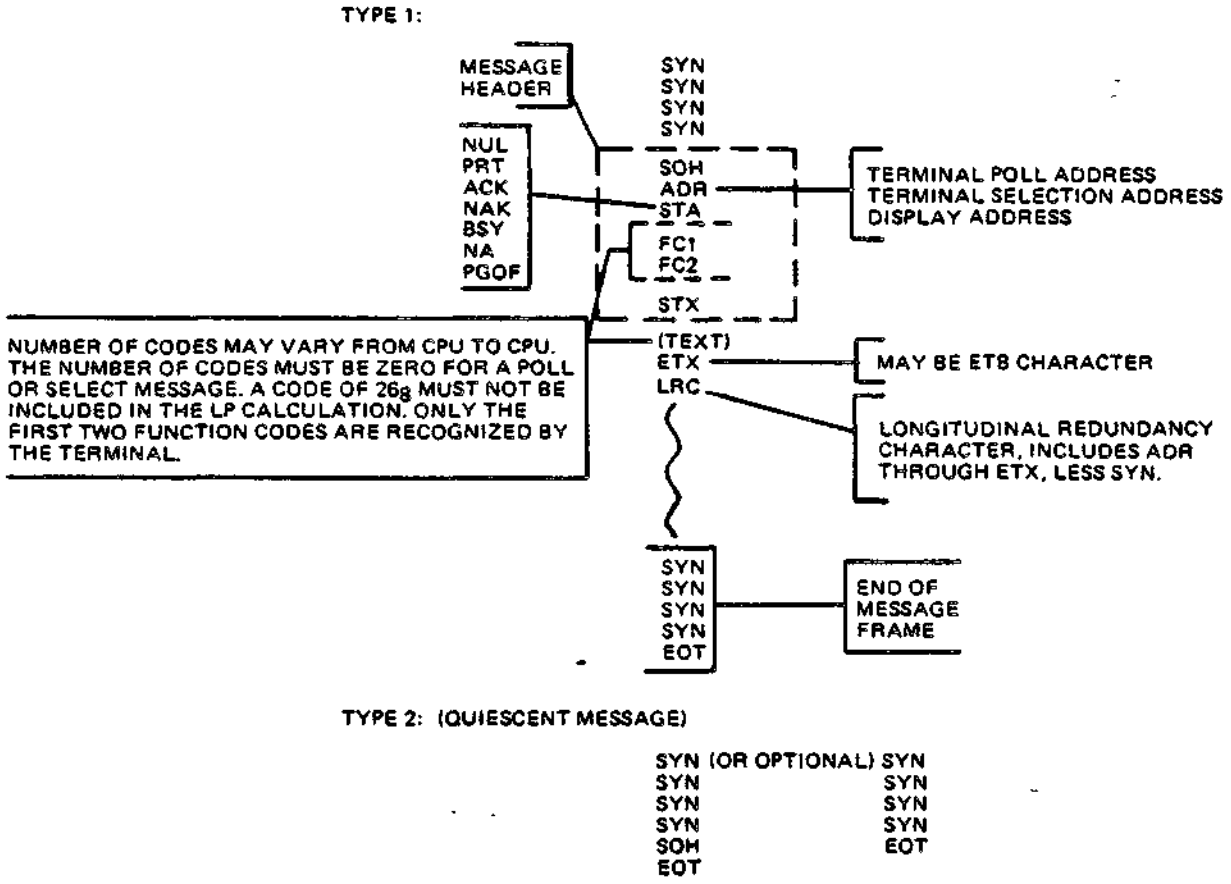


Figure 10-3. VIP Protocol message Structure for PVE

PVE HARDWARE FUNCTION CODES

PVE hardware function codes are listed in the appropriate hardware device manuals.

These codes provide a special message-labeling capability to be used by the application.

The application can include two function codes in the message header of each text message by setting at connect time the following in the IORB: (1) set to 1, bit 8 of the device-specific word I_DVS (see Table 10-2); and (2) set to 1, bit B (extension bit) of I_CT2 to specifiy that the IORB is extended (see Figure 4-2 and Table 4-11). The line protocol handler then inserts the two user-specified hardware function codes at read time into the IORB's I_FCS item.

PVE INPUT DATA

The line protocol handler places into the application's buffer all data between the STX and ETX control characters. The data is inserted into the buffer in 7-bit ASCII, with the most significant bit always zero. The LPH strips the ETX and LRC (longitudinal redundancy check character, see Section 7, "Communications Subsystem Error and Correction Procedures") from the data and does not include them in the buffer.

It also strips DEL characters when the application, at connect time, sets to 1 the A-bit of the device-specific word I_DVS (Table 10-2).

By setting the E- and F-bits of I_DVS as shown in Table 10-2, the application can control the response that the line protocol handler sends when it receives data, but no read IORB is available.

Control and Characteristics of PVE Output

PVE OUTPUT MESSAGE HEADER

The PVE line protocol handler normally supplies the output header between the SOH and STX control characters. The application can specify hardware function codes (1, 2) as described above under "PVE Hardware Function Codes." To write function codes, the application program must, at connect time, set bit B (extension bit) of the IORB's I_CT2 item to 1, to specify that the IORB is extended. At write time, when specified, the codes are extracted from the I_FCS item of the IORB. When the codes are not specified (bit 8 of I_DVS set to 0 at connect time), the line protocol handler will supply two spaces, instead of the codes, into I_FCS. (See Figure 4-2 and Table 4-11.)

PVE TERMINAL ADDRESS (ADR) AND MESSAGE STATUS (STA)

The PVE line protocol handler supplies an ADR (terminal address) of X'60' (keyboard/screen) and an STA (message status) of NUL to the application.

PVE OUTPUT DATA

The application's output data must be 7-bit ASCII. The most significant bit is used by the line protocol handler during transmission of odd parity.

Output data must not include the ASCII control characters SOH, STX, ETB, ETX, EOT, or SYN.

The line protocol handler supplies output ETX control characters and longitudinal redundancy check characters (LRCs) (described in Section 7, "Communications Subsystem Error and Correction Procedures").

PVE LINE PROTOCOL HANDLER TIMEOUT INTERVALS

Table 10-4 lists the timeout intervals used by the line protocol handler for the connect, read, and write functions. The line protocol handler will attempt or reattempt the functions until the indicated timeout period has elapsed.

In addition to the interval in the table, there is also a gross timeout of one minute, which expires when the control station ceases to poll or select any tributary station.

Table 10-4. PVE Timeout Intervals

Function	Timeout Interval
Connect	200 seconds
Read	Indefinite
Write	Indefinite

ERROR REPORTING BY PVE LINE PROTOCOL HANDLER

Table 10-5 lists the errors reported by the PVE line protocol handler. It also lists corresponding return status error codes (see Table 4-10) and corresponding bits in the software status word I_ST (see Table 10-3).

Table 10-5. Errors Reported by PVE Line Protocol Handler

Error Condition	Posted Error Return Status	I_ST Bit	Comments
No interrupt from MLCP	6	7	Poll failure or CCP/MLCP failure
NAK limit reached	7	8	Write failure
Purged due to immediate close	B	None	
Station disabled	B	None	
Fatal error interrupt level	B	None	
Data service rate error	0 (send) 7 (receive)	2 2, 8	Not fatal
Communication control block service rate error	7	4, 8	
Long record	0	6	Not fatal
Phone hang-up	B	B	
Nonexistent resource, or Bus parity error, or Unrecoverable memory error	B	None	

Section 11
2780/3780 BSC LINE PROTOCOL HANDLER

BSC 2780/3780 LINE PROTOCOL HANDLER

The binary synchronous transmission (BSC) 2780/3780 line protocol handler (LPH) supports BSC 2780 and BSC 3780 point-to-point, nontransparent or transparent EBCDIC, or non-transparent ASCII transmission between a DPS 6/Level 6 system and another host system (subject to certain restrictions).

The 3780 protocol is similar to the standard 2780 protocol and unless specifically stated otherwise, the rest of this section and the term BSC pertain to both.

GENERAL BSC LINE PROTOCOL HANDLER OPERATION

When a station (device or computer) at either end of a communication line has a message to send, it requests use of the line by sending an ENQ bit message. (See Appendix G for definition of ENQ and other control characters.) The receiving station must respond with an ACK/0 sequence before the sending station can transmit a data message.

BSC Transmit and Receive Operations

A station that has control of the line, i.e., the right to transmit, is known as the master (primary) station. The station that relinquishes control, i.e., will receive, is the slave (secondary) station. Primary and secondary are arguments of the BSC CLM directive used during system build.

When the first data message from the master station is successfully received, the slave station responds with an ACK/1 sequence. Acknowledgments for subsequent remaining messages alternate between ACK/0 and ACK/1. The master/slave status for each respective station remains in effect until the master station gives up control by sending an end-of-transmission (EOT) character (which is not acknowledged by the slave station).

When a bidding station does not receive an ACK/0 response within a specified interval (timeout period), it sends another ENQ message. At the same time, or at nearly the same time, the other station may be sending an ENQ message, bidding for the line. Thus both stations may be bidding with neither receiving an ACK response. This is known as line contention. Line contention can be avoided by designating one station as the primary and the other as secondary during system build. Then when the designated primary station receives an ENQ response to its bid message, it retransmits the ENQ message to the secondary station, which in turn ignores its own bid request and responds to the primary station with an ACK or NAK.

The BSC line protocol handler allows a receiving station to reply to a data message with an reverse interrupt (RVI) message if it has an urgent requirement to transmit data.

Figure 11-1 illustrates bids and other interactions between a master and slave station.

BSC Data Transmission Modes

BSC operates in either basic data transmission mode or in advanced data transmission mode, according to whether a control byte is included in the data being transmitted. (See "BSC Control Byte (Receive)" and "BSC Control Byte (Send)" later in this section.)

BSC BASIC DATA TRANSMISSION MODE

In basic data transmission mode, there is no control byte included in the data being transmitted along the communications line.

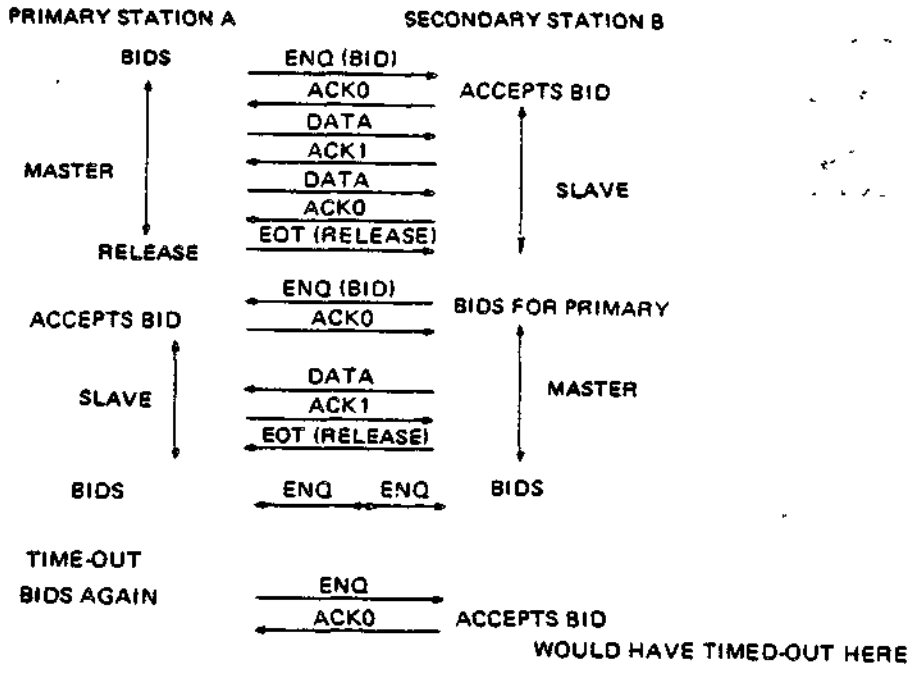


Figure 11-1. Example of BSC Communication

BSC ADVANCED DATA TRANSMISSION MODE

In advanced data transmission mode, the application includes a control byte that occupies the first byte of the output buffer but is not transmitted across the line. The control byte indirectly controls the operation of the line protocol handler (e.g., sending an ETB or ETX), or conveys information about a data transfer (e.g., whether transparent text was received).

BSC 2780 and BSC 3780 Differences

The 3780 protocol differs from the 2780 protocol in that the 3780 protocol allows an application to:

- Receive a conversational reply
- Receive two records and to transmit a single record, when the double-block option is selected at connect time (whereas the 2780 protocol allows both transmission and reception of two records)
- Receive multi-block records and to transmit a single record, when the multi-block option is selected at connect time (whereas the 2780 protocol allows both transmission and reception of multi-block records)
- Receive and transmit selected BSC control characters in nontransparent mode.

BSC Record Types

The BSC LPH supports three forms of record transmission:

1. Single-record transmission
2. Two-buffer transmission
3. Multiple intermediate text block (ITB) sequence.

To identify the record constructs in a more meaningful and uniform manner, the following terms will be used:

- Single-block (in place of single-record or single-buffer)
- Double-block (in place of two-buffer)
- Multi-block (in place of multiple ITB sequences).

BSC 2780/3780 Features

The following discussions in this subsection include references to BSC-specific fields in the input/output request block IORB (see Table 4-8) and to control bytes. See Tables 11-4 and 11-5 later in this section for descriptions of the device-specific word I_DVS and software status word I_ST, respectively. Control bytes are described under "Control Byte (Receive)" and "Control Byte (Transmit)".

BSC DOUBLE-BLOCK FEATURE

With the double-block feature, the use of the second buffer reduces line turnaround time, i.e., two records can be transmitted with only one acknowledgment. However, there are these disadvantages:

- When a line (parity) error occurs, both records must be retransmitted.
- One transmission requires that two writes be issued, which are not posted until an acknowledgment is received.
- Four buffers are necessary to operate the line efficiently.

Figure 11-2 shows record transmissions with and without the double-block feature.

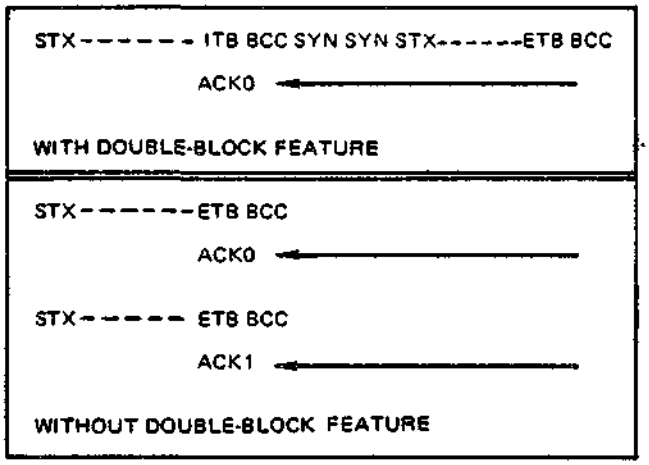


Figure 11-2. BSC Double-block Feature in Record Transmission

Before selecting the double-block feature, compare the advantage of better line utilization against the disadvantages of a more complex program and increased buffer usage, and consider the following:

1. In BSC 2780 with the double-block option, two records can be received or transmitted (using an ITB (intermediate text block) sequence).

2. In BSC 3780, with the double-block option, two records can be received, using an ITB sequence, and single records can be transmitted. This implies that an application using BSC 3780 must be able to receive up to two records at any one time, but can only initiate single-record transmission.

3. The double-block feature cannot be used with synchronous reads, because the intermediate files being received may be terminated by an ETX record. If the ETX record is the first of the two records being read, the second read (synchronous) would not be posted to the system.

   For example:

```
READ (asynchronous)
.
. process
.
READ (synchronous)
.
.  process
.
```

Assumes always two records per transmission.

The following sequence is better:

```
READ (asynchronous)
READ (asynchronous)
WAIT (1)
.
.  process
.
READ (asynchronous)
WAIT (2)
.
.  process
.
```

BSC MULTI-BLOCK FEATURE

The multi-block feature allows an application to send or receive from 1 to 7 records in a single transmission. Use of multi-block reduces line turnaround time in that only one write order, one user buffer, and one acknowledgment are required for the transmission of multiple records.

This feature is optionally selected at connect time. When using BSC 2780, an application selects the multi-block feature to both send and receive multiple-record transmissions for the duration of the connect; single- and double-block transmissions are precluded. When using BSC 3780, an application selects the multi-block feature only to receive multi-block transmissions.

For this feature to be selected at connect time, its use must have been provided for at system build. This is accomplished by an argument in the BSC CLM directive. Indicating the possible use of the multi-block feature during system build does not require that it be selected for use at connect time, since selection is optional. However, selection of this feature at connect time is prohibited if the possibility of its use was not provided for during system build.

When 3780 and multi-block are specified at connect time, the receive buffer must be organized as shown in Figure 11-3. Transmit buffer organization remains the same as for single-record transmission. When 2780 and multi-block are specified at connect time, the receive and transmit buffers must be organized as shown in Figure 11-3. The buffer shown in Figure 11-3 is divided into two sections, a header section and a data section. The data section contains the user's records, referred to as data blocks. Only the data blocks of a data buffer are transmitted, with the appropriate protocols inserted. The header section is interpreted by and controls the processing of the BSC LPH. Table 11-1 defines the contents of the buffer's header section. Figure 11-4 illustrates the transmission of the data blocks shown in Figure 11-3.

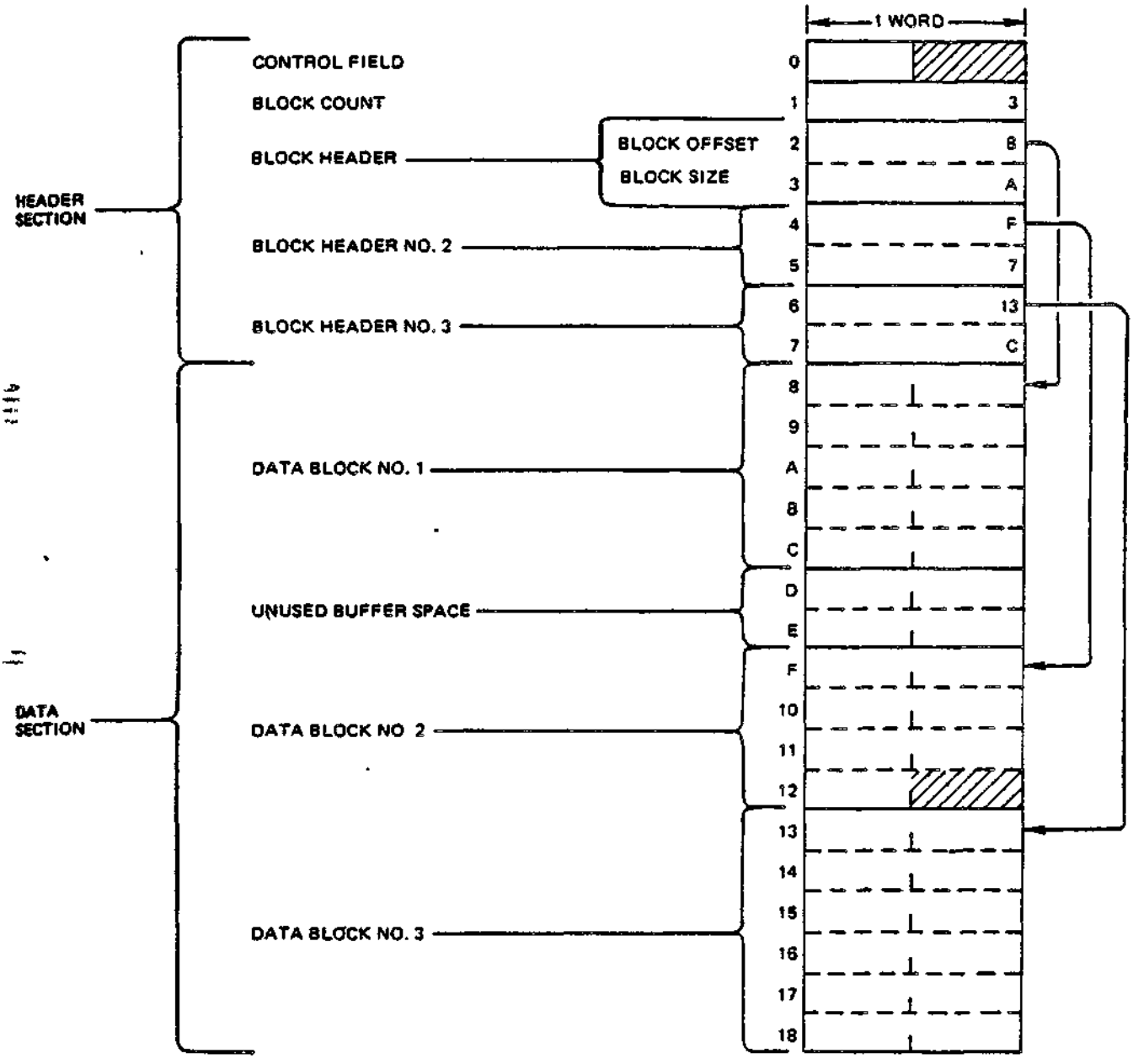


Figure 11-3. Multi-block Buffer Organization

Table 11-1. Multi-block Header Section Field Descriptions

	Transmit (2780 Only)	Receive (2780/3780)
Control Field	Byte 0: Contains optional control byte. Byte 1: Must be zero.	Same as for transmit.
Block Count	Number of blocks to be transmitted. Posted back with actual number transmitted.	Maximum number of blocks which can be received. Posted back with actual number received.
Block Offset	Word offset from base of buffer to beginning of data block. Posted back with contents unchanged.	Same as for transmit.
Block Size	Number of characters (bytes) in data block to be transmitted. Posted back with residual range, if any.	Maximum number of characters (bytes) which can be received. Posted back with actual number received.

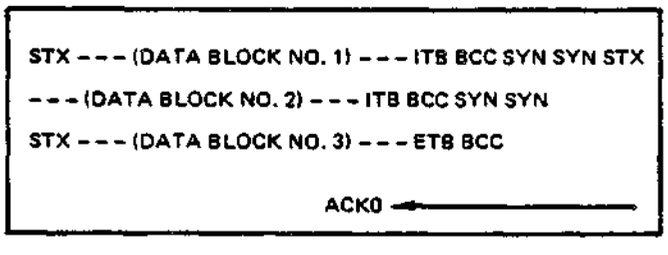


Figure 11-4. BSC Multi-block Transmission of Buffer Shown in Figure 11-3.

The following rules apply to the construction of a multi-block buffer:

- Data blocks cannot overlap.
- Each data block begins on a word boundary.
- The range value in the IORB need not be specified; this value is calculated from the buffer header section by system software. The range value returned in the posted read IORB is the header size plus total block lengths plus any gap(s) between blocks. Range value returned in the posted write IORB is equal to the header size.
- Buffer space may exist between data blocks but must not be used because its contents may be overwritten by system software.
- Block headers and the corresponding data blocks they define must be in the same sequence. The first header block must define the first data block, the second header block must define the second data block, etc. These header blocks precede the data blocks.
- During a single connect, the number of data blocks can vary for each transmission, provided they do not exceed the maximum allowed. This maximum number, which can vary from 1 to 7, is specified by the user at system build.
- If, during a read operation, fewer data blocks are received than were specified in the block-count field of the buffer's header section, the block-count field is set equal to the number of data blocks received and the IORB is posted back with bit A of the software status word I_ST set to 1, indicating nonzero residual range.

BSC TEMPORARY TEXT DELAY (TTD) FEATURE

The following describes the sequence of the temporary text delay (TTD) feature:

1. When a master station receives an ACK, and no output request block (IORBs) are queued, that station waits 2 seconds for one IORB (or two IORBs when there are two buffers) to be queued.
2. The master station then sends the temporary text delay (TTD) control character sequence (STX, ENQ) to the slave station.

3. When the slave station responds with a NAK, the master station checks whether the application has queued the appropriate write requests. If the write requests are not queued, the master station continues the TTD sequence until the application issues the necessary write requests.

4. If the EOT or ETX bit (A-bit or D-bit) in the I_DVS word of the IORB is set (Table 11-4), one write request will effect transmission.

Figure 11-5 is an example of the temporary text delay sequence.

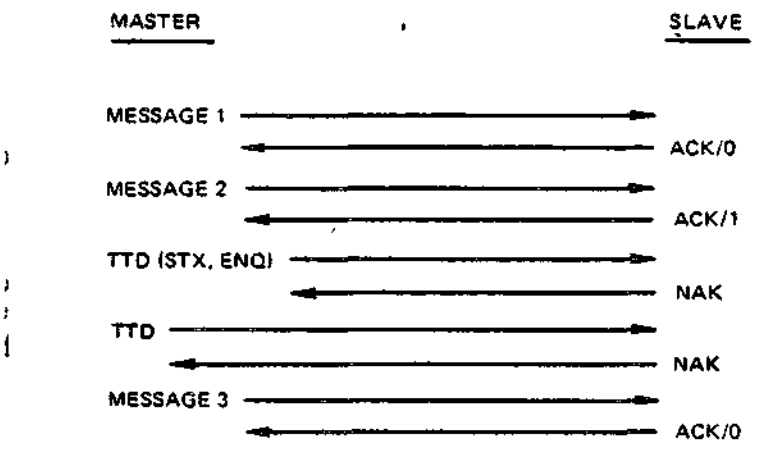


Figure 11-5. BSC Temporary Text Delay (TTD) Sequence Example

BSC WAIT BEFORE ACKNOWLEDGE (WACK) FEATURE

A BSC slave station will send ACK/0 and ACK/1 responses to messages satisfactorily received, provided there is at least one outstanding read request (two with the double-block feature), in addition to the request being processed.

1. When no read is queued, the slave station posts the current read, waits 2 seconds for read requests to be queued, then sends a WACK response, indicating to the master station that the last message was received, but the slave station cannot accept more data.

2. The master station waits (timeout), then sends an ENQ message.

3. If a read request was queued during the timeout, the slave station responds with an ACK, and the master station can send its next data message.

4. If no read request was queued during the timeout, the slave station waits another 2 seconds, and when necessary sends another WACK sequence.

The ASCII and EBCDIC WACK sequences are DLE ; and DLE , respectively.

Figure 11-6 is an example of the wait before acknowledge (WACK) sequence.

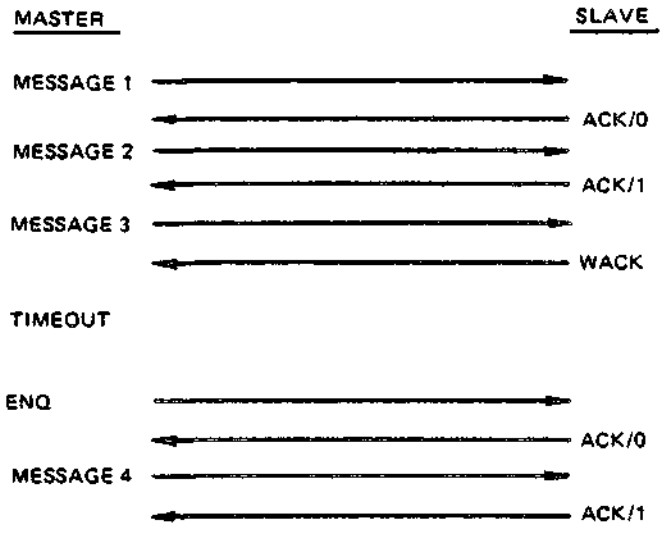


Figure 11-6. BSC Wait Before Acknowledge (WACK) Sequence Example

BSC REVERSE INTERRUPT (RVI) FEATURE

When a slave station is processing read requests and must unexpectedly transmit an urgent message, that station must issue a reverse interrupt (RVI) message, which informs the master station that the slave station is requesting control of the line.

On receiving an RVI character, the master station should empty its buffers and give up control of the line. However, the master station does not have to acknowledge the RVI by giving up control.

The application program can request the BSC line protocol handler to send an RVI character, by either of the following methods:

1. Use of the control byte. The application issuing read requests issues a transmit request with bit 5 of the control byte set to 1 (see Figure 11-12) and with the urgent message in the application's buffer.

2. Use of the device-specific word I_DVS of the IORB. The application issuing read requests issues a transmit request with the B-bit of I_DVS set to 1 and with the urgent message in the application's buffer.

The application issuing write requests can detect an RVI character by either of these methods:

1. Test bit 3 of the control byte after a successful write request is posted. A bit setting of 1 indicates that the RVI for that IORB was received.

2. Test bit 3 of the IORB's software status word I_ST. A bit setting of 1 indicates an RVI was received.

Figure 11-7 is an example of a reverse interrupt (RVI) sequence.

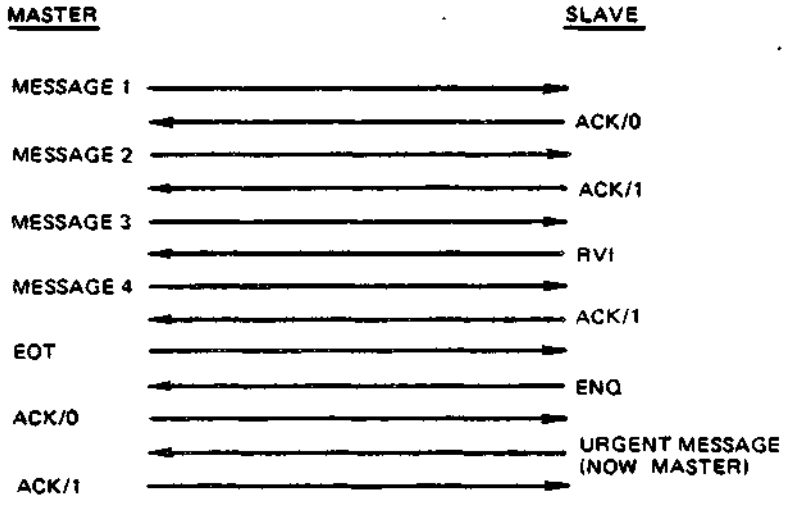


Figure 11-7. BSC Reverse Interrupt (RVI) Sequence Example

BSC END OF TRANSMISSION (EOT) FEATURE

The application program, by any one of the following methods, can cause the BSC line protocol handler to send an end-of-transmission (EOT) message:

- At connect time, specify use of the control byte by setting to 0 bit 4 of the IORB's device-specific word I_DVS. When bit 4 of the first byte of the application's buffer (control byte, specified at write time) is set to 1, the BSC line protocol handler will send an EOT control character after the data in the application's buffer is successfully transmitted.

- When the control byte is not specified at connect time, set to 1 A-bit of the IORB's device-specific word I_DVS at write time. The BSC line protocol handler will send an EOT control character after the data in the application's buffer is successfully transmitted.

- After successful completion of a write request, issue a disconnect with or without a queue abort, and no physical disconnect. The master station will send an EOT character and give up its master status. However, when another IORB is queued for write, that station will again request its master status.

The application can detect receipt of an EOT control character in either of the following ways:

- If the control byte was specified at connect time, bit 4 of the control byte, of the read request on which the EOT was received, will be set to 1.

- If the control byte was not specified at connect time, bit 12 of the software status word I_ST, of the request on which the EOT character was received, will be set to 1.

With either method, the line protocol handler does not post any read requests queued before the EOT character was detected. To remove read requests from the queue, the application must issue a disconnect with a queue abort. The line protocol handler always posts the IORB with a device unavailable (B) return status (Table 4-10). The BSC line may or may not be available for further use, depending on whether or not an EOT character was sent abnormally.

BSC SWITCHED LINE DISCONNECT (DLE EOT) FEATURE

A DLE EOT sequence is used to indicate the imminent intent of the transmitting station to do a physical disconnect. Use of this feature is selected at connect time by setting bit 5 of the IORB's I_DVS word to 1. If bit 5 is instead set to 0, the line protocol handler will transmit EOT instead of a DLE EOT. This transmission of the EOT will occur as described in the preceding description of EOT.

Reception and notification of a DLE EOT sequence, indicating pending line hang-up by the transmitting station, is performed in two ways:

- If the control byte was not specified at connect time, bits 9 and C are both set to 1 in the IORB's software status word for the read request on which the DLE EOT was received.

- If the control byte was specified at connect time, bits 3 and 4 of the control byte are both set to 1 for the read request on which the DLE EOT was received.

Transmission of a DLE EOT sequence is initiated for a disconnect request when the following two conditions are both true:

- Bit F of the disconnect IROB's device-status word is set to 0 (this is a request for a physical disconnect).

- Bit 5 of the connect IORB's device-specific word was set to 1.

Table 11-2 defines the conditions under which EOT or DLE EOT is selected for transmission or reported as having been received.

Table 11-31. Transmission and Reception Conditions for EOT and DLE EOT

Receiving Station		
Connect IORB Selected	Character Transmitted	Reported to Application
EOT	DLE EOT	EOT
DLE EOT	DLE EOT	DLE EOT
DLE EOT	EOT	EOT

Transmitting Station		
Connect IORB Selected	Disconnect IORB Specified	Character Transmitted
EOT	Logical Hangup	EOT
EOT	Physical Hangup	EOT
DLE EOT	Logical Hangup	EOT
DLE EOT	Physical Hangup	DLE EOT

BSC Line Protocol Handler Timeout Interval

When a line is idle (no station controls the line), the timeout interval in waiting for a line-request bid is 10 minutes.

Once a station has successfully bid for a line, the timeout interval for subsequent reads (from the slave station) or writes (from the master station) is 12 seconds.

BSC Features Specific to 3780

BSC 3780 CONVERSATIONAL REPLY FEATURE

The conversational reply feature permits a 3780 application, after transmission of an entire message (whose last record is denoted by an ETX rather than an ETB), to selectively receive a message from a host computer without a preliminary line bid sequence.

The conversational reply sequence serves as the affirmative reply to the last message transmission block, and as a break or interrupt to later transmissions. The line protocol handler indicates to the application receipt of a conversational reply sequence in bit 5 of the IORB software status word I_ST, and/or in bit 2 of the control byte of the ETX write order.

In the following example, a 3780 application attempts to transmit three 2-record messages to a remote host computer. The transmission sequence is interrupted by the receipt of a conversational reply, which occurs after transmission of the second message. After the complete conversational reply (containing one or more records) is received, transmission of the third message can resume, following completion of a successful line bid sequence. Figure 11-8 illustrates the example sequence.

The application's use of the conversational reply feature requires that the application issue the requisite number of read orders (dependent on single or double-block mode) before the transmission of a text block that terminates with an ETX sequence. If the application does not issue the required read(s), the last text block is not transmitted, and the line protocol handler will initiate a temporary text delay (TTD) sequence until the necessary read orders are issued. If the application does not transmit an ETX sequence, it need not issue supporting read order(s).

BSC 3780 DOUBLE-BLOCK FEATURE

The discussion under "BSC Double-block Feature" earlier in this subsection applies also to BSC 3780 operation.

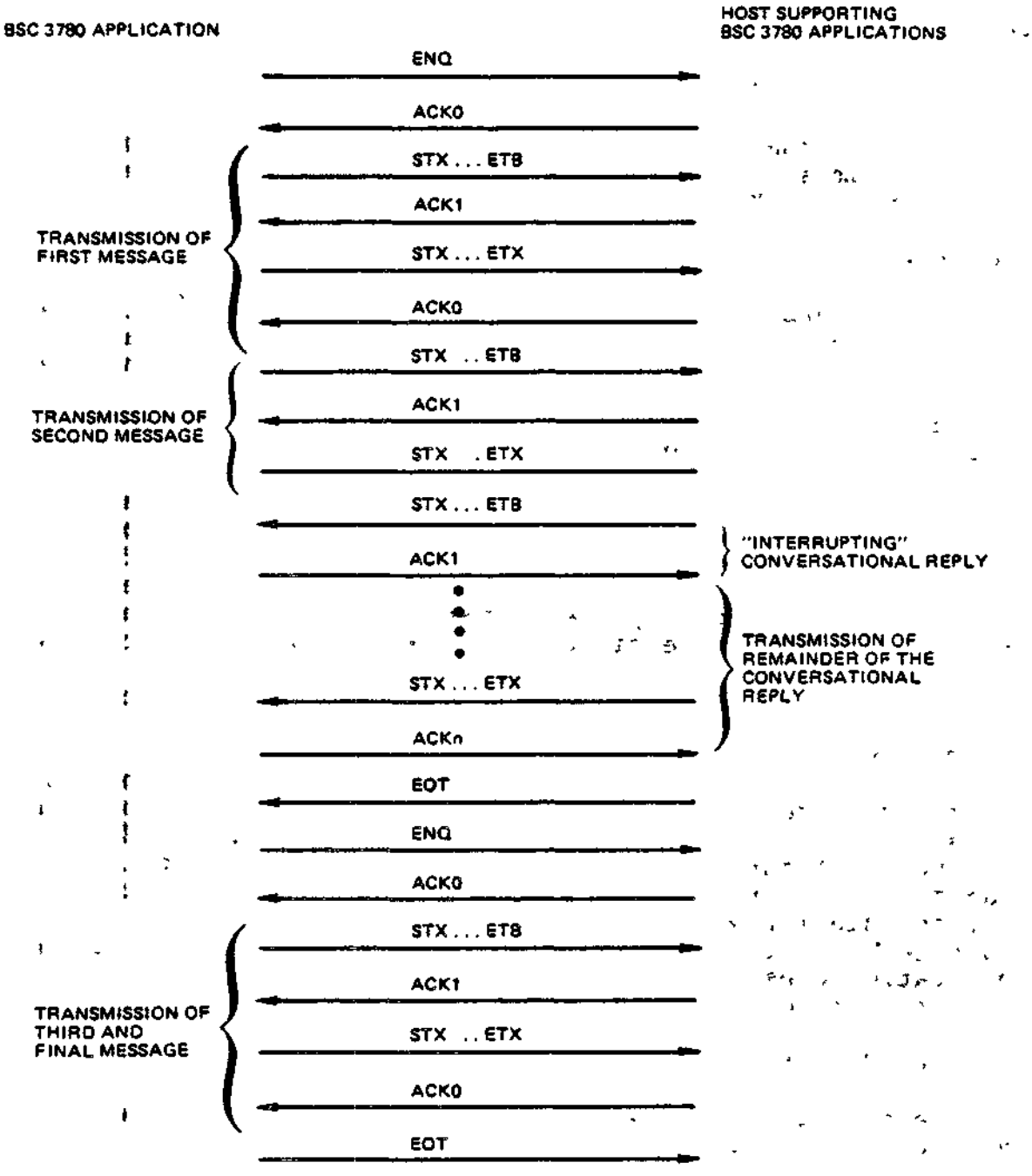


Figure 11-8. Example of Conversational Reply in BSC 3780 Transmission Sequence

BSC 3780 TRANSMISSION/RECEPTION OF BSC CONTROL CHARACTERS

In BSC 2780 nontransparent mode, detection of any BSC control characters within a message would abort the transmission or reception of that message.

In 3780 nontransparent mode, selected, noncritical BSC control characters (i.e., STX, SOH, DLE, NAK, and EOT) can be successfully transmitted and received.

USING THE BSC 2780/3780 LINE PROTOCOL HANDLER

BSC-Specific IORB Values

The BSC-specific IORB item I_CT2, device-specific word I_DVS, and software status word I_ST, are shown and defined in Tables 11-3, 11-4, and 11-5, respectively. User-specified bits not specifically described in the tables must be 0. Section 4 has a general description of the IORB.

Table 11-3. Function Codes in I_CT2 Field in the IORB

Function Code	Definition	Use
0	Wait online	Used by the line protocol handler to complete the description of the requested I/O function.
1	Write	
2	Read	
A	Connect	
B	Disconnect	

Table 11-4. BSC Device-Specific Word I_DVS in the IORB

Bit Number	Bit Setting	Meaning of Bit Setting
0	0	Must be zero.
1	0	Must be zero.
For connect call only (function code A)		
2	0	Do not use Auto Call Unit.
	1	Use Auto Call Unit.
3	0	Must be zero.
4	0	Use control byte.
	1	Do not use control byte.
5	0	Do not support DLE EOT seqeuence.
	1	Support DLE EOT sequence.
6	0	Must be zero.
7	0	Use single-block or double-block feature. (See bit 8.)
	1	Use multi-block feature. For 2780: Both send and receive. For 3780: For receive only. (Bit 8 must be zero.)
		NOTE Bit 7 must be zero for bit 8 to be meaningful.
8	0	Use single-block per transfer.
	1	For 2780: Use double-block for send/receive. For 3780: Use double-block for receive.
9	0	Use BSC 2780 protocol.
	1	Use BSC 3780 protocol.

Table 11-4 (cont). BSC Device-Specific Word I_DVS in the IORB

Bit Number	Bit Setting	Meaning of Bit Setting
For write call only (function code 1)		
A	0	Do not send EOT after this transmission.
	1	Send EOT after this transmission.
B	0	Do not send RVI if station is in slave status.
	1	Send RVI if station is in slave status.
C	0	Send data in nontransparent mode.
	1	Send data in EBCDIC transparent Mode.
D	0	Send ITB or ETB charcters following the data.
	1	Send ETX characters following the data.
For disconnect call only (function code B)		
E	0	Abort (dequeue) all IROBs on request queue.
	1	Process outstanding requests on request queue.
F	0	Disconnect line on completion. If bit 5 was set to 1 on connect, then send DLE EOT sequence.
	1	Do not disconnect line on completion.

Specifying Use of BSC 2780 and/or 3780 to the System

The inclusion of BSC 2780 and/or 3780 in the system is done during system build. The application can select and use either 2780 or 3780 according to the setting of bit 9 in the device-specific word I_DVS in the IORB (see Table 11-4).

Table 11-5. BSC Software Status Word I_ST in the IORB

Bit	Meaning When Bit Set to 1
0	N/A
1	N/A
2	Data service rate error
3	Lost line bid; RVI received
4	Communications control block service error
5	Conversational reply received (3780 only)
6	Long record
7	0 = ITB and/or ETB characters received 1 = ETX character received
8	N/A
9	0 = not meaningful 1 = DLE EOT received, if bit C is also 1
A	Nonzero residual range
B	Phone hang-up
C	EOT character received
D	Transparent message received
E	NAK limit reached
F	Fatal error: bus parity or memory error

Formats and Characteristics of BSC Input Data

The formats and characteristics of BSC input data for both ASCII and EBCDIC are described and illustrated below.

Figure 11-9 shows the format and contents of BSC input data received from another computer.

SOM (START OF MESSAGE)
A ONE- OR TWO-CHARACTER SEQUENCE THAT IS STRIPPED BY THE BSC LPH.

CONTROL BYTE
THE CONTROL BYTE, IF SPECIFIED, IS THE FIRST BYTE OF THE APPLICATION'S DATA

DATA
INFORMATION STORED IN THE APPLICATION'S BUFFER AND SPECIFIED AT READ TIME.

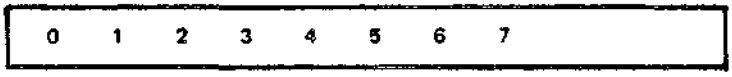
EOM (END OF MESSAGE)
A ONE- OR TWO-CHARACTER SEQUENCE THAT IS STRIPPED BY THE BSC LPH.

BCC
AN LRC CHARACTER OR CRC CHARACTER THAT IS INSERTED BY THE BSC LPH.

Figure 11-9. BSC Input Data Format and Contents

BSC CONTROL BYTE (RECEIVE)

When bit 4 of the IORB's device-specific word I_DVS is set to 0 at connect time (see Table 11-4), the BSC line protocol handler uses the first byte of the application's buffer as the control byte. Figure 11-10 shows the control byte's format and content.



BITS 0 THROUGH 2
NOT APPLICABLE; NOT EXAMINED

BIT 3 = 0
NOT MEANINGFUL

BIT 3 = 1
DLE EOT RECEIVED IF BIT 4 IS ALSO 1

BIT 4 = 0
DATA STORED IN APPLICATION'S BUFFER

BIT 4 = 1
EOT RECEIVED; NO DATA STORED IN APPLICATION'S BUFFER

BIT 5
NOT APPLICABLE; NOT EXAMINED

BIT 6 = 0
DATA RECEIVED IN NONTRANSPARENT MODE

BIT 6 = 1
DATA RECEIVED IN TRANSPARENT MODE

BIT 7 = 0
ITB OR ETB RECEIVED

BIT 7 = 1
ETX RECEIVED

Figure 11-10. Control Byte (Receive) for BSC Line Protocol Handler

ASCII INPUT FOR BSC

ASCII input characteristics and format (Figure 11-9) are as follows:

1. SOM (start-of-message) consists of the STX control character only.

2. The control byte (if specified at connect time) is stored in the first byte of the application's buffer, and indicates the end-of-message (EOM) sequence. When bit 7 is 0, it indicates detection of an ITB or ETB control character; when 1, it indicates detection of an ETX character. Note that bit 7 of both the control byte and of I_ST are specified.

3. Data must be 7-bit ASCII with odd parity. The BSC line protocol handler strips the parity bit and resets it to zero when it stores it in the application's buffer.

4. The EOM sequence, one of the three control characters ITB, ETB, or ETX, is indicated by bit 7 of the IORB software status word I_ST after a successful read is posted. See Table 11-5 for bit 7 indicators.

5. The BCC (block check character) is described in Section 7, "Line Protocol Handler Functions."

EBCDIC INPUT FOR BSC

EBCDIC input format and characteristics are as follows:

1. SOM (start-of-message) consists of the STX control character only.

2. The control byte (if specified at connect time) is stored in the first byte of the application's buffer, and indicates the end-of-message (EOM) sequence, as follows:

   Bit 4 = 1 End of transmission (EOT) detected.
   Bit 7 = 0 ITB or ETB character detected.
   Bit 7 = 1 ETX character detected.

3. Data must be 8-bit EBCDIC; it will not have any BSC control characters.

4. The EOM sequence, one of the control characters ITB, ETB, or ETX, is indicated by bit 7 of the IORB software status word I_ST after a successful read is posted. See Table 11-5 for bit 7 indicators.

5. The BCC (block check character) is described in Section 7, "Line Protocol Handler Functions."

TRANSPARENT EBCDIC INPUT FOR BSC

Transparent EBCDIC input format and characterisitcs are as follows:

1. SOM (start-of-message) consists of the two-character sequence DLE, STX.

2. The control byte, if specified at connect time, is stored in the first byte of the application's buffer, and indicates the EOM (end-of-message) sequence according to the bit 7 setting (Figure 11-10).

3. Data may be any EBCDIC character, including BSC control characters.

4. EOM (end-of-message) sequence may be one of the following, indicated by bit settings of the IORB software status word I_ST, after a successful read has been posted:

I_ST Bits

D	Z	Resulting EOM Sequence
1	0	DLE, ITB
1	0	DLE, ETB
1	1	DLE, ETX

5. The block check character (BCC) is described in Section 7, "Line Protocol Handler Functions."

Formats and Characteristics of BSC Output Data

Formats and characteristics of BSC output data (both ASCII and EBCDIC) are described and illustrated below.

Figure 11-11 shows the format and content of BSC data transmitted to another computer.

SOM
A ONE- OR TWO-CHARACTER SEQUENCE THAT IS INSERTED IN FRONT OF THE DATA BY THE BSC LPH.

CONTROL BYTE
THE CONTROL BYTE, IF SPECIFIED, IS STORED IN THE FIRST BYTE OF THE APPLICATION'S BUFFER.

EOM
A ONE- OR TWO-CHARACTER SEQUENCE THAT IS INSERTED BY THE BSC LPH.

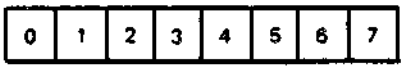
BCC
AN LRC CHARACTER OR CRC CHARACTER THAT IS INSERTED BY THE BSC LPH.

DATA
INFORMATION THAT IS TRANSMITTED FROM THE APPLICATION'S BUFFER BY THE BSC LPH.

Figure 11-11. Format and Content of BSC Output

BSC CONTROL BYTE (SEND)

When bit 4 of the IORB's device-specific word I_DVS is set to 0 at connect time (see Table 11-14), the BSC line control handler uses the first byte of the application's buffer as the control byte. Figure 11-12 shows the format and content of the BSC line protocol handler's control byte for sending data.



BITS 0, 1
NOT APPLICABLE, NOT USED
BIT 2=1
CONVERSATIONAL REPLY RECEIVED
BIT 3=1
RVI RECEIVED (RETURN STATUS ONLY)
BIT 4=1
SEND THE DATA THAT IS IN YOUR BUFFER AND, AFTER IT HAS BEEN ACKNOWLEDGED, SEND EOT
BIT 5=1
SEND AN RVI RESPONSE ON THE NEXT ACKNOWLEDGMENT OF A READ
BIT 6=0
SEND NONTRANSPARENT EBCDIC
BIT 6=1
SEND TRANSPARENT EBCDIC OR ASCII
BIT 7=0
SEND ITB OR ETB
BIT 7=1
SEND ETX

Figure 11-12. Control Byte (Send) for BSC Line Protocol Handler

BSC ASCII OUTPUT

ASCII output characteristics and format are as follows:

1. SOM (start-of-message) consists of only the STX character.

2. The control byte, when specified, is assumed to be the first byte of the application's buffer, and indicates the EOM (end-of-message) sequence, which is either ITB, ETB, or ETX, designated as follows:

   a. Bit 6 must be 0.

   b. Bit 7 = 0. Send ITB or ETB. ITB is sent when the record is odd numbered (1, 3, 5, etc.) and the double-block feature is used.

      Bit 7 = 1. Send ETX.

   If the control byte is not specified, the EOM sequence is defined by I_DVS as described in 4 below.

3. Data must be 7-bit ASCII; it cannot have any BSC control characters.

4. EOM, which is either ITB, ETB, or ETX, can be indicated by the control byte (see 2 above) or by the C- and D-bits of the IORB device-specific word I_DVS (Table 11-4 as follows):

   a. C-bit must be zero.

   b. D-bit = 0. Send ITB or ETB. ITB is sent when the record is odd-numbered (1, 3, 5, etc.) and the double-block feature is used.

      D-bit = 1. Send ETX.

5. BCC (block check character) is described in Section 7, "Line Protocol Handler Functions."

BSC EBCDIC OUTPUT

EBCDIC output characteristics and format are as follows:

1. SOM (start-of-message) consists of only the STX character.

2. The control byte, when specified, is assumed to be the first byte of the application's buffer, and indicates the EOM (end-of-message) sequence, which is either ITB, ETB, or ETX, designated as follows:

a. Bit 6 must be 0.

b. Bit 7 = 0. Send ITB or ETB. ITB is sent when the record is odd-numbered (1, 3, 5, etc.) and the double-block feature is used.

Bit 7 = 1. Send ETX.

If the control byte is not specified, the EOM sequence is defined by I_DVS as described in 4 below.

3. Data may be 8-bit EBCDIC; it cannot have any BSC control characters.

4. EOM (end-of-message), which is either ITB, ETB, or ETX, can be indicated by the control byte (see 2 above) or by the C- and D-bits of the IORB device-specific word I_DVS (Table 11-11) as follows:

a. C-bit must be zero.

b. D-bit = 0. Send ITB or ETB. ITB is sent when the record is odd-numbered (1, 3, 5, etc.) and the double-block feature is used.

D-bit = 1. Send ETX.

5. BCC (block check character) is described under "Line Protocol Handler Functions", earlier.

BSC TRANSPARENT EBCDIC OUTPUT

Transparent EBCDIC output characteristics and format are as follows:

1. SOM (start-of-message) consists of the two-character sequence DLE, STX.

2. The control byte, when specified, is assumed to be the first byte of the application's buffer, and indicates the EOM (end-of-message) sequence, which is either DLE ITB, DLE ETB, or DLE ETX, designated as follows:

a. Bit 6 must be 0.

b. Bit 7 = 0. Send DLE ITB or DLE ETB. DLE ITB is sent when the record is odd-numbered (1, 3, 5, etc.) and the double-block feature is used.

Bit 7 = 1. Send DLE ETX.

If the control byte is not specified, the EOM sequence is defined by I_DVS as described in 4 below.

3. Data may be any EBCDIC character, including any BSC control characters.

4. EOM, which can be either DLE ITB, DLE ETB, or DLE ETX, can be indicated by the control byte (see 2 above) or by bit 4 and bit D of the IORB device-specific word I_DVS (Table 11-4) as follows:

   a. Bit 4 must be 1.

   b. D-bit = 0. Send DLE ITB or DLE ETB. DLE ITB is sent when the record is odd-numbered (1, 3, 5, etc.) and the double-block feature is used.

      D-bit = 1. Send DLE ETX.

5. BCC (block check character) is described in Section 7, "Line Protocol Handler Functions".

Section 12
TTY LINE PROTOCOL HANDLER

TTY LINE PROTOCOL HANDLER

The TTY line protocol handler supports asynchronous terminal devices, generically classified as teleprinter-compatible (TTY), that include certain ASR, KSR, and visual information projection (VIP) terminals.

A basic TTY terminal consists of either a printer and keyboard or a VIP7100/7200/7800 display and keyboard. (Paper tape is not supported.) Each type of TTY terminal has an asynchronous communications interface that permits operation at up to 9600 baud.

GENERAL TTY LINE PROTOCOL HANDLER OPERATION

TTY Message Formats

Figure 12-1 illustrates TTY message formats. On input, the application receives only the text portion of the message. On output messages, the application can control print format with a control byte that is specified as the first character of the output buffer (in the IORB device-specific word I_DVS, described later). At connect, read, or write, the application can, with the I_DVS word, dynamically specify which message format is to be used.

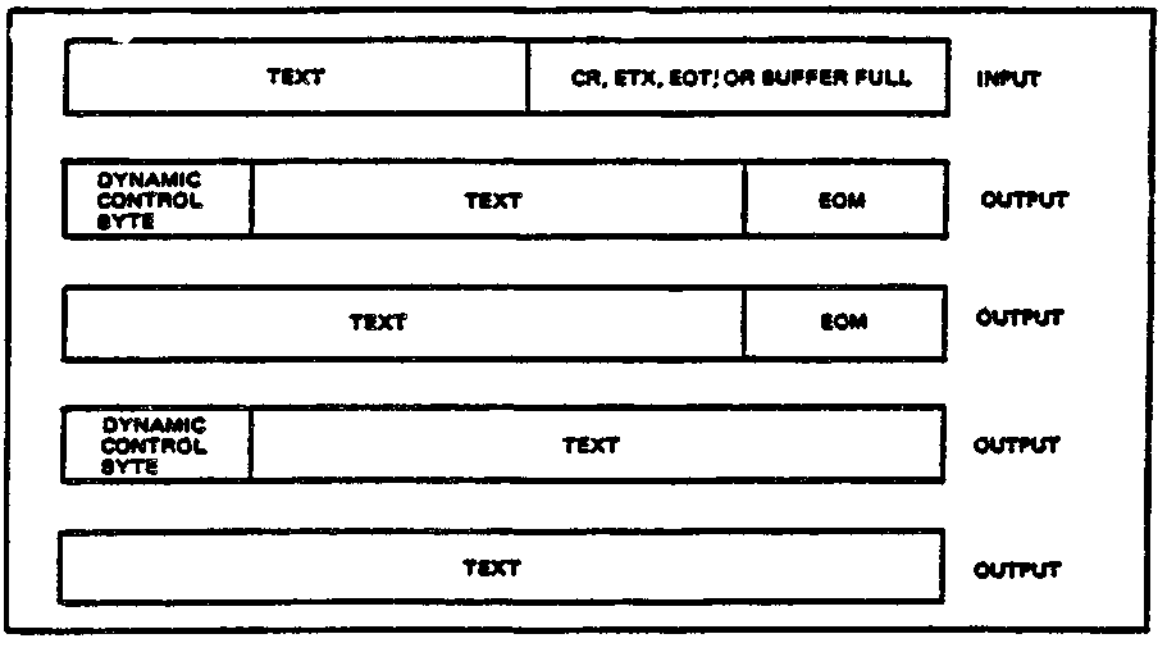


Figure 12-1. TTY Message Formats

TTY Character Mode and Buffered Mode Transmission

TTY CHARACTER MODE

Transmission for all TTY terminals is usually in character mode (one character at a time), a characteristic of the hardware that provides that:

- The TTY line protocol handler does all editing of data before any transmission.
- Multiple input lines are not allowed at the same time.

TTY BUFFERED MODE (VIP7200 AND VIP7800)

For VIP7200 and VIP7800 only, the buffered mode, available as a hardware option, permits:

- The TTY line protocol handler to process multiple lines of input at the same time
- The operator to do local editing of data before it is transmitted
- The application to instruct the TTY line protocol handler not to edit input data.

Buffered mode permits the TTY line protocol handler to process a write order while a read order is pending. A "quasi full duplex" operation gives the line protocol handler the ability to have the application send to the terminal sequences that cause the terminal to send information back to the application's buffer.

Buffered quasi full duplex operates as follows:

1. When the channel control program (CCP) of the multiline communications processor (MLCP) is currently processing a write order to the terminal, a subsequent read or write operation is not given to the CCP until the current write order completes.

2. When the CCP is processing a read order and the next order is a write order, that write order is processed while the read order is active.

3. When the write order (step 2) completes and the read order has not yet completed, a subsequent read or write order will not be processed until the read is completed. When the read order is completed before the write order, actions in 1 above take effect.

4. When the read order is completed, the line protocol handler returns to its original state, i.e., no orders pending. The line protocol handler can initiate read or write orders to the CCP.

VIP7200 AND VIP7800 HARDWARE SWITCH OPTIONS WITH CHARACTER OR BUFFERED MODE

The TTY line protocol handler supports the following VIP7200/VIP7800 hardware switch options for character mode or buffered mode operation as follows:

Character Mode	Buffered Mode
CHARACTER/BUFFER switch in CHARACTER position	CHARACTER/BUFFER switch in BUFFER position
Internal Even/Odd Parity switch set to select EVEN	Internal Even/Odd Parity switch set to select EVEN
HALF/FULL DUPLEX switch in FULL position	HALF/FULL DUPLEX switch in FULL position
	LINE/PAGE switch as required by user
	Internal end-of-message switch set to select ETX or EOT only

VIP7200 AND VIP7800 FUNCTION AND CONTROL KEYS

Function and control keys on the VIP7200 and VIP7800 are supported only in buffered mode.

When issuing a write request that will cause an automatic response by the terminal, the application must first issue an asynchronous read request, then issue a write request that contains a control message to the terminal.

TTY Line Protocol Handler Timeout Intervals

Table 12-1 lists the TTY line protocol handler's timeout intervals for the LPH functions.

Table 12-1. TTY Line Protocol Handler Timeout Intervals

Line Protocol Handler Function	Timeout Interval
Connect	Five minutes
Read	Character mode: five minutes after receipt of the first character of the message
	Buffered mode: five minutes after the line protocol handler receives the request.
Write	Thirty seconds

USING THE TTY LINE PROTOCOL HANDLER

TTY-Specific IORB Values

The TTY-specific IORB item I_CT2, device-specific word I_DVS, and software status word I_ST are shown and defined in Tables 12-2, 12-3, and 12-4, respectively. User-specified bits not specifically described in these tables must be 0. Section 4 describes the general form of the IORB.

Table 12-2. Function Codes in I_CT2 of the IORB

Function Code	Definition	Use
0 1 2 A B	Wait online Write Read Connect Disconnect	Used by the line protocol handler to complete the description of the requested I/O function

Table 12-3. TTY Device-Specific Word I_DVS in the IORB

Bit Number	Bit Setting	Meaning of Bit Setting
0	0	Must be zero.
1	0	Must be zero.
For connect call only (function code A)		
2	0	Do not use Auto Call Unit.
	1	Use Auto Call Unit.
3	0	Must be zero.
4	0	First byte in buffer on output is a control byte.
	1	First byte in buffer on output is a data byte.
For read call only (function code 2)		
5	0	Input data is in nontransparent mode.
	1	Input data is in transparent mode.
6	0	Must be zero.

Table 12-3 (cont). TTY Device-Specific Word I_DVS in the IORB

Bit Number	Bit Setting	Meaning of Bit Setting
For write call only (function code 1)		
7	0	Stop output immediately on detecting a BRK received from the terminal.
	1	Continue output when BRK detected.
8	0	Must be zero.
9	0	Must be zero.
For read call only (function code 2)		
A	0	Do not echo keyboard input.
	1	Echo keyboard input.
For read and write calls (function codes 2, 1)		
B	0	No LF (line feed) at end of message.
	1	LF (line feed) at end of message.
C	0	CR (carriage return) at end of message.
	1	No CR (carriage return) at end of message.
For connect call only (function code A)		
D	0	Data transfer is in character mode.
	1	Data transfer is in buffered (block) mode.
For disconnect call (function code A)		
E	0	Abort (dequeue) all IORBs on the request queue.
	1	Process outstanding requests on the request queue.
F	0	Hang up phone after disconnect.
	1	Do not hang up phone after disconnect.

Table 12-4. TTY Software Status Word I_ST in the IORB

Bit	Meaning When Bit Set to 1
0	N/A
1	N/A
2	Data service rate error
3	N/A
4	Communications control block (CCB) service error
5	No stop bit in character input
6	Long record
7	N/A
8	N/A
9	N/A
A	Nonzero residual range
B	Phone hang-up
C	N/A
D	N/A
E	N/A
F	Fatal error: bus parity or memory error

Control and Characteristics of TTY Input Data

This subsection describes user control over the characteristics of TTY input data, and applies to character-mode processing unless otherwise noted.

TTY CONTROL BYTE (INPUT)

The description of the TTY control byte for output (see "TTY Control Byte (Send)" below) applies also to the TTY line protocol handler's control byte for input.

TTY NONTRANSPARENT INPUT

TTY input is nontransparent when the application sets to 0 bit 5 of the IORB's device-specific word I_DVS (Table 12-3). Input is accepted until the end-of-range or a CR (carriage return), ETX (end of text), or EOT (end of transmission) control character, whichever is first, is reached. The line protocol handler does not transmit the CR, ETX, or EOT control character as part of the message.

TTY TRANSPARENT INPUT

TTY input text is transparent when the application sets to 1 bit 5 of the device-specific word I_DVS at read time (Table 12-3). All input data, including any control characters, is stored in the buffer until end-of-range is reached.

TTY LINE FEED (LF) AND CARRIAGE RETURN (CR) INPUT SEQUENCE

The application can specify at read time a sequence of LF and CR characters, with the B- and C-bits of the IORB's device-specific word I_DVS, as indicated in Table 12-3. When the message is received successfully, the specified character combinations are retransmitted back to the terminal.

KEYBOARD INPUT CHARACTER AND LINE CONTROL

When an input character with a parity error is received, the line protocol handler sends a BEL character back to the terminal. The user must then retype that input character if it is to be included in the text being sent to the application.

The user can correct or delete erroneous characters or lines and can declare control characters to be data characters, as described below.

To correct one or more characters in the current line, i.e., before the CR is pressed, press the @ key. This deletes the character that immediately preceded the @ character, and displays the @ symbol. Each succeeding @ entry deletes another character, moving from right to left to the beginning of the line.

To delete the current line, i.e., before the CR is entered, press and hold the CTRL (control) key and press X. This deletes the current line, displays the message *DEL* on the next line, and results in a carriage return. The user can then enter a correct line.

To cause a control character (e.g., @, CTRL X, CR, and \) to be accepted as a data character (transparent mode) press the backslash (\) key before entering that control character. The system interprets the backslash as an escape character. In transparent mode, all input characters are data characters and have no editing functions.

TTY DISPLAY OF INPUT CHARACTERS

The user can cause an input character to be echoed to the terminal (displayed on the screen or typed on the console) by setting to 1 the A-bit of the device-specific word I_DVS (Table 12-3). For full duplex printers, the application need specify that characters be returned only when they are to be echoed by the system software.

TTY INPUT IN BUFFERED MODE (VIP7200 AND VIP7800 ONLY)

When the application at connect time sets to 1 the D-bit of the device-specific word I_DVS, input is accepted until an ETX or EOT control character or end-of-range is encountered.

When the application sets bit 5 of I_DVS to 1 at read time, TTY input in buffered mode is transparent, i.e., there is no editing. When the bit 5 is set to 0, TTY input in buffered mode is nontransparent; i.e., control characters are edited.

As in character mode, the application can specify an LF and CR sequence, as described above under "Line Feed (LF) and Carriage Return (CR) Input Sequence."

Control and Characteristics of TTY Output Data

This subsection describes user control of the characteristics of TTY output data and is applicable to character-mode processing unless otherwise stated.

TTY CONTROL BYTE (SEND)

The TTY line protocol handler's control byte, included as the first character of the application's buffer, controls the message's head-of-form sequence. At connect time, the application specifies the control byte by setting to 0 bit 4 of the IORB's device-specific word I_DVS (Table 12-3).

Figure 12-2 shows the format and content of the TTY control byte.

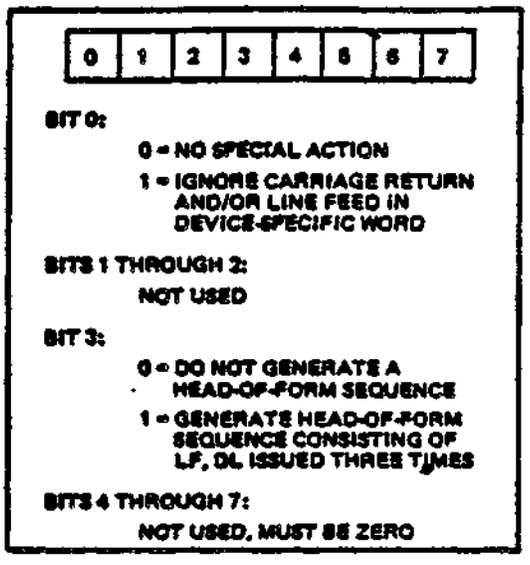


Figure 12-2. Control Byte for TTY Line Protocol Handler

END-OF-MESSAGE (EOM) SEQUENCE ON TTY OUTPUT

The EOM sequence is controlled by the B- and C-bits of the IORB's device-specific word I_DVS (Table 12-3), as specified by the application at write time. The TTY line protocol handler sends an EOM sequence according to the following B- and C-bit values:

I_DVS Bits		
B	C	EOM Sequence
0	0	CR, DEL, DEL
0	1	DEL, DEL
1	0	LF, CR, DEL, DEL
1	1	LF, DEL, DEL

At read time, the application can specify the same B- and C-bit values in order to send an EOM sequence back to the terminal when the message is successfully received.

TTY DETECTION OF BRK CHARACTERS

When the application sets to 0 bit 7 of the device-specific word I_DVS at write time, the line protocol handler will immediately stop all output when it detects a BRK key character in the input stream from the terminal. The line protocol handler ignores the BRK character when bit 7 is set to 1, until the write order is completed.

TTY OUTPUT IN BUFFERED MODE

Control and characteristics for TTY output in buffered mode are the same as described above for character mode. However, in processing in buffered mode (VIP7200/7800 only), the line protocol handler processes all physical I/O requests in the same sequence as they are received. If there is already an outstanding read request, only a subsequent write request can be initiated before the read request is satisfied or the timeout for that read request is elapsed.

Section 13
SYSTEM ACCESS

This section describes MOD 400 user access procedures.

USER ACCESS PROCEDURES

When you are at a user terminal, access to the system depends on the way your terminal is described to and recognized by the system.

Access to the system requires:

1. Physical connection between your terminal and the central processor.

2. Logical connection between you (the user) and the operating system.

In some cases, the Executive performs the second step for you automatically after you have made the physical connection.

CONNECTING THE TERMINAL TO THE CENTRAL PROCESSOR

You can connect your terminal to the central processor by two methods, depending on the type of terminal you have: a direct-connect terminal or a dialup terminal.

logged in, your access to system facilities is governed by control arguments entered in the LOGIN command or, under user registration, in your user profile.

MANUAL LOGIN TERMINAL

When the connection to the system has been made at a manual login terminal, a message-of-the-day and the login prompter message:

```
LOGIN
```

followed by the system identification and the current date and time appear at the terminal. If the terminal (and your user profile) allow it, you may enter a full login line, such as:

```
L JONES
```

to gain access to the system. (Check with the system operator for the correct format of your full login line.) In a user registration system, you may then be required to enter your password in response to the prompter message:

```
PASSWORD?
```

If you enter the login line and password correctly, the system responds with a ready message and you can begin to enter commands.

ABBREVIATED LOGIN TERMINAL

Some terminals and some user profiles allow login only by abbreviation. Most systems have defined one or more abbreviations that are available to users at any terminal, and may have defined terminal-specific abbreviations in addition. Check with the system administrator for the abbreviations that can be used at your terminal.

If you wish to login by an abbreviation after the login prompter message has been issued, enter a one-character login abbreviation, such as:

```
S
```

In a user registration system, you may then be required to enter your password in response to the prompter message:

```
PASSWORD?
```

If you have entered the correct abbreviation and password, the system will respond with a ready message and you can begin to enter commands.

To find out what files are under this directory, enter:

```
LS
```

and the system responds with a listing of the files. Figure 13-1 shows a sample listing.

PROCEDURES AND CONVENTIONS AFTER ACCESS

It may be necessary to request operator intervention or interrupt a running task while at your terminal. The procedures and conventions used to perform these actions are described in the following paragraphs.

Sending Messages to the Operator

To send a message from your terminal to the system operator, you can enter the Message (MSG) command described under "Working With Files". For example, if you want to abort the current batch request, you could enter:

```
MSG "PLEASE ABORT BATCH REQUEST"
```

Interrupting (Breaking) a Task

You can interrupt or break a running task to reenter commands, temporarily halt the task, or terminate it.

To effect a break from the user terminal, press the BREAK (BRK) key. The system then issues the break prompter message:

```
**BREAK**
```

Your response may be any one of the following:

1. Enter any command (see the Commands manual). This may be followed by another command or by one of the responses described in steps 2 through 4. If the entered command is not Start (SR), Logoff (BYE), New Procedure (NEW_PROC), Unwind (UW), or Program Interrupt (PI) (described later), the lead task again enters break mode and issues another **BREAK** message requesting another response.

2. Enter one of the following break mode responses to the **BREAK** message:

   a. Start (SR). This resumes execution of the suspended task as though the break had not been made.

   b. Unwind (UW). This releases all tasks and you return to command level.

   c. Logoff (BYE). This aborts and deletes the current task group request.

   d. New Process (NEW_PROC). This aborts all task requests in the task group except for the lead task, then restarts the task group, using the same arguments as specified in the initial task group request.

   Any of these commands terminates the current break; i.e., there will be no other **BREAK** message after they are executed.

3. Enter Unwind (UW). All tasks will be terminated and you return to command level.

   If the terminated task was invoked following a break, the lead task reenters break mode, issues another **BREAK** prompter message, and awaits a response.

4. Enter Program Interrupt (PI). The task interrupted is currently suspended.

   For Linker and Editor, suppress output and return to directive input level. The PI command suppresses output resulting only from the Linker MAP directive.

   The PI command is meaningful only to the Linker and Editor running in a task group whose lead task is the Command Processor. The commands described in steps 1, 2, and 3 may be used with the Linker and Editor.

Example:

You issue a List Names (LS) command and the output begins to appear on the screen at your terminal. You want this output to be printed on the line printer. You should immediately press the Break (BRK) key and take one of the following steps:

Section 14
FILE CONVENTIONS

This section presents MOD 400 file conventions as well as a procedural scenario titled "Working With Files". This scenario provides a detailed explanation of frequently used file system commands and procedures.

OVERVIEW

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 to identify and locate files have been established for their effective control when 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, and printers) also use the file concepts. However, since unit record devices cannot be used to store files, there is less need to

The following paragraphs describe the root directory and other special types of directories.

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 14-1 is a root directory.

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, is required at all times. This volume normally contains system programs, commands, and other routinely used elements.

SYSTEM BOOT DIRECTORY

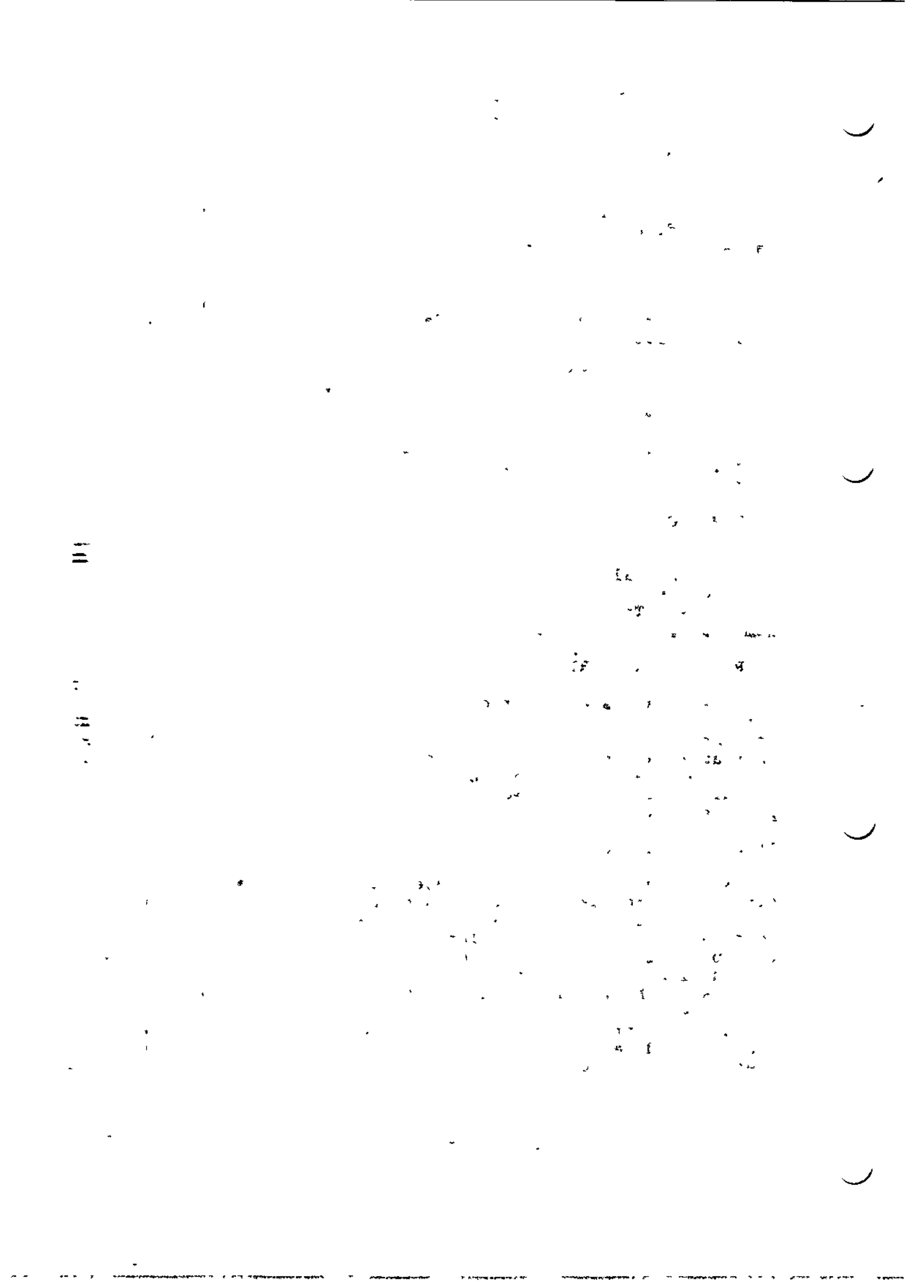
The system boot directory is the root directory of the volume used to initialize the system. It must contain a number of directories and files that the system needs to perform its functions. These are described in the System Building and Administration manual.

USER ROOT DIRECTORIES

The File System can recognize one or more user root directories. These are root directories of volumes 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 unit files, listing files, or anything else that you want to store, either temporarily or permanently.

INTERMEDIATE DIRECTORIES

When a volume is first created, it contains only a root directory. You can create, within this directory, 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 of these people has one or more files of interest to him. 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.



path of the working directory is made known to the File System, and if the desired element is contained in that directory, the element can be specified by just its name. The File System 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 on a volume a directory or file resides. 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. Hence, when a new element is created, it is allocated the first available space. If more space is needed, it will be obtained from another free area. Thus, there is not necessarily any relationship between a file's extents and contiguous free disk sectors.

Naming Conventions

Each disk file and directory name in the File System can consist of the following ASCII characters: uppercase alphabetics (A through Z), digits (0 through 9), underscore (_), hyphen (-), dollar sign ($), and period(.). If lowercase alphabetic characters are used, they are converted to their uppercase counterparts.

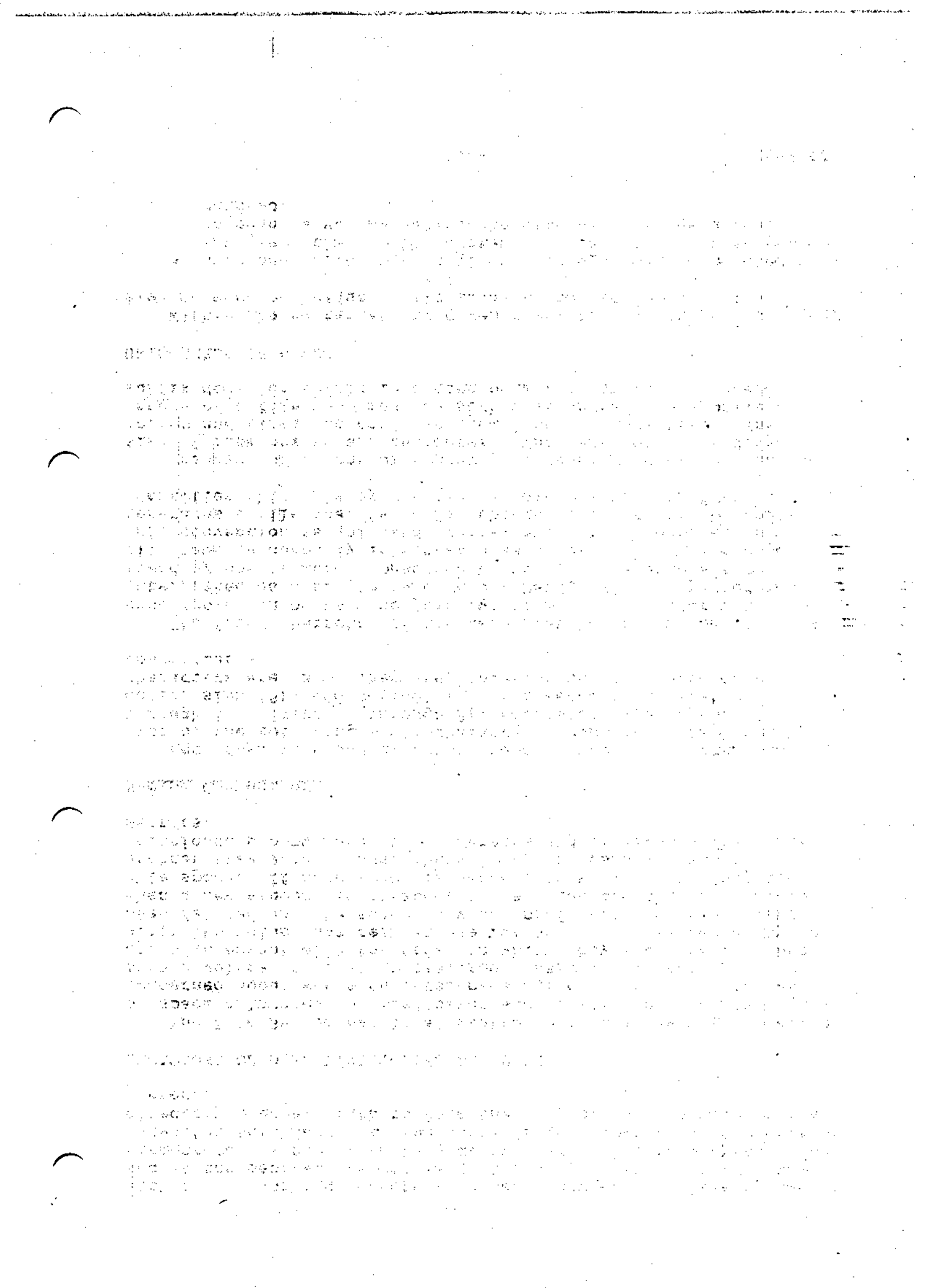
The first character of any name must be an alphabetic. The underscore can be used to join two or more words that are to be interpreted as a single name (e.g., DATE_TIME). A period followed by one or more alphabetic or numeric characters after a file name is normally interpreted as a suffix to a file 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 comprise 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:

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



- 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 a 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 file name, depending on the function to be performed. In the Create Directory command, for example, 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.

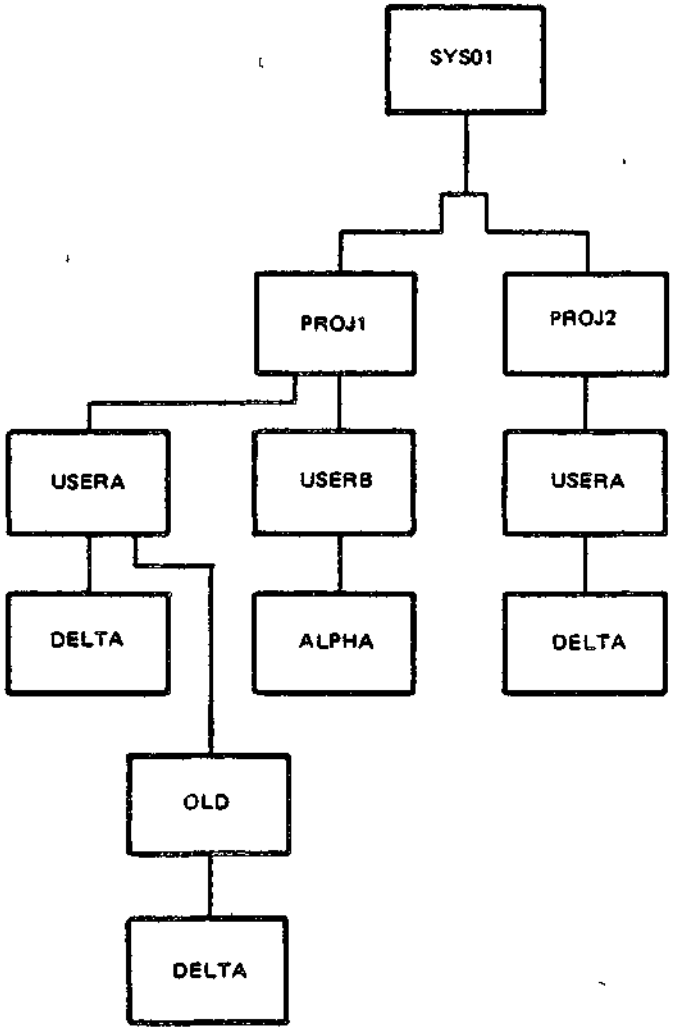
Absolute and Relative Pathnames

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 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.

Absolute Pathname. An absolute pathname is one that begins with a circumflex (^) or a greater-than symbol (>). (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 volume, the volume from which the system was initialized, is mounted.)

When an absolute pathname begins with a greater-than symbol, the first element named in the pathname is assumed to be immediately subordinate to the system volume root directory. Thus, if the system volume name is SYS01 and the pathname given is >DIR1>FILEA, the full pathname becomes ^SYS01>DIR1>FILEA.

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 14-3. Sample Pathnames

- Plus sign (+)
- Comma (,)
- Hyphen (-)
- Period (.)
- Slash (/)
- Colon (:)
- Semicolon (;)
- Less-than sign (<)
- Equal sign (=)
- Question mark (?)
- Underscore (_).

The underscore (_) 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 these characters can be used as the first character of a file or volume name.

The name of a tape volume can be from one through six characters; 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. Therefore, 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 files).

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 the tape volume accessible to the File System software.

UNIT-RECORD DEVICE FILE CONVENTIONS

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

A command function reads its own input during execution from the user-in file (normally assigned to your terminal). The directives submitted to the Line Editor following entry of the Editor command, for example, are submitted through user-in. A task group normally writes its output to the user-out file (normally assigned to your terminal). The user-out file can be reassigned to another device or file (see "Controlling Output"). This reassignment remains in effect until another reassignment occurs.

The Command Processor, and any commands it invokes, writes any errors detected to the error-out file. The error-out file is the same as the initial user-out file; it cannot be reassigned by a command or command argument.

You can determine the full pathnames associated with each of these files by issuing a Status Group (STG) command at your terminal.

Command Level

The system indicates that it is at command level by issuing a ready (RDY) message at your terminal. This assumes that you have not disabled the ready message by a previously issued Ready Off (RDF) command; if you have, the system still comes to command level, but you are not informed. You can activate the ready prompt at any time by issuing a Ready On (RDN) command.

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

- After a command function terminates, the system returns to command level and awaits the entry of another command. This command can be any function you wish to execute or it can be a BYE command, indicating that you have no further work to do and you want to terminate the current session.

- You can interrupt execution of an invoked command by pressing the Break or Interrupt key at your terminal. See "Interrupting Execution" below.

CONTROLLING YOUR OPERATING ENVIRONMENT

The following paragraphs describe the commands and procedures that you may find most useful as an interactive system user. Once at command level, you can perform a wide variety of system operations using these commands and special system procedures. Selected examples are designed to help you become familiar with using the system for applications programming. For full descriptions of all commands and their arguments, refer to the Commands manual.

You can now use the Create Volume (CV) command to assign a unique vol_id to your new disk volume, using the following form of the command:

```
CV !DSK00 -FT WORK
```

where WORK is the vol_id you want to assign.

Using the -FT argument initializes all data structures on the volume and establishes WORK as the root directory name; the root directory pathname for this volume is ^WORK.

RENAMING DISK VOLUMES

If disk volumes having the same vol_id are used, one of the volumes must be renamed before the system will accept it. (A tape volume cannot be renamed.) The command:

```
CV !DSK00>OLD -RN NEW
```

renames the volume OLD using the -RN control argument; the new volume name is NEW.

Directory Control

You can create an unlimited number of directories to organize your files. The following commands illustrate how to change your working directory, and create, rename, or delete directories.

CHANGING YOUR WORKING DIRECTORY

The system enables you to keep aware of your location within the directory and file structure at any moment. You can also request a list of the files and directories under any directory to which you have list access.

To list your working directory, use the List Working Directory (LWD) command:

```
LWD
^SYSVLA>UDD>PROGS>LOWELL
```

The system responds with the absolute pathname of your working directory. If you want to change to some other directory, use the CWD (Change Working Directory) command. For example:

```
CWD ^SYSVLA>UDD>PROGS>JONES
RDY:
LWD
^SYSVLA>UDD>PROGS>JONES
```

The simple pathname of your new working directory is JONES. Any number of users can work in the same directory at one time.

Before creating your two directories, you enter a CWD command to change your working directory to ^WORK:

```
CWD ^WORK
```

(Note that this step is optional; you need not change your working directory to the volume ^WORK to create subordinate directories or files. You can create directories or files from any location in the File System tree structure by supplying the appropriate absolute or relative pathname of the file or directory you wish to create. However, for the sake of simplicity, only simple pathnames are used here.)

To create the directory SHEPHERD, enter the command:

```
CD SHEPHERD
```

This directory now resides immediately subordinate to the root directory ^WORK.

To create the directory COOK, enter the command:

```
CD COOK
```

This directory now resides, along with SHEPHERD, immediately subordinate to the root directory ^WORK. Figure 14-4 illustrates this directory tree structure.

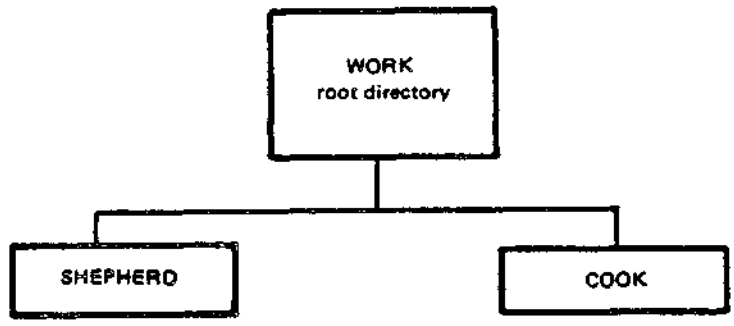


Figure 14-4. Location of Directories SHEPHERD and COOK

RENAMING DIRECTORIES

You can change the name of an existing directory using the Rename (RN) command. For example, assume that within your working directory >UDD>PROGS>SMITH, there is a directory TEST. The command:

```
RN TEST WORK
```

changes the pathname of the affected directory from:

```
>UDD>PROGS>SMITH>TEST to >UDD>PROGS>SMITH>WORK
```

As another example, assume that you wish to create a file under each of the two directories, SHEPHERD and COOK, shown in Figure 14-5. Your working directory is the root directory WORK. To create a file named REPORTS under the directory SHEPHERD, enter the command:

```
CF SHEPHERD>REPORTS
```

where SHEPHERD>REPORTS is the relative pathname (relative to your working directory) of the file you wish to create.

The file REPORTS now resides immediately subordinate to the directory SHEPHERD, as shown in Figure 14-5.

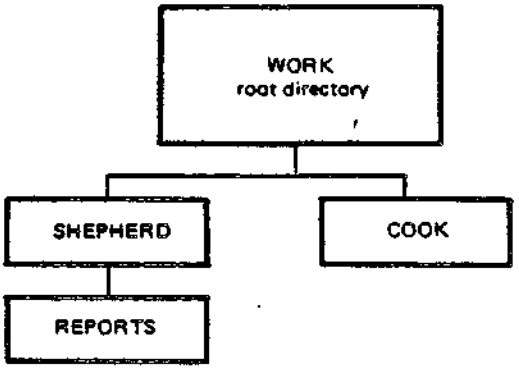


Figure 14-5. Location of Subordinate File REPORTS

Suppose you want to create a file named WORDLIST under the directory COOK. Since your working directory is still the root directory, WORK, enter the command:

```
CF COOK>WORDLIST
```

where COOK>WORDLIST is the relative pathname of the file you want to create. The file WORDLIST now resides immediately subordinate to the directory COOK, as shown in Figure 14-6.

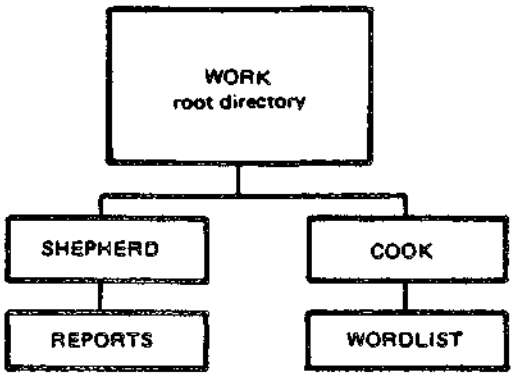
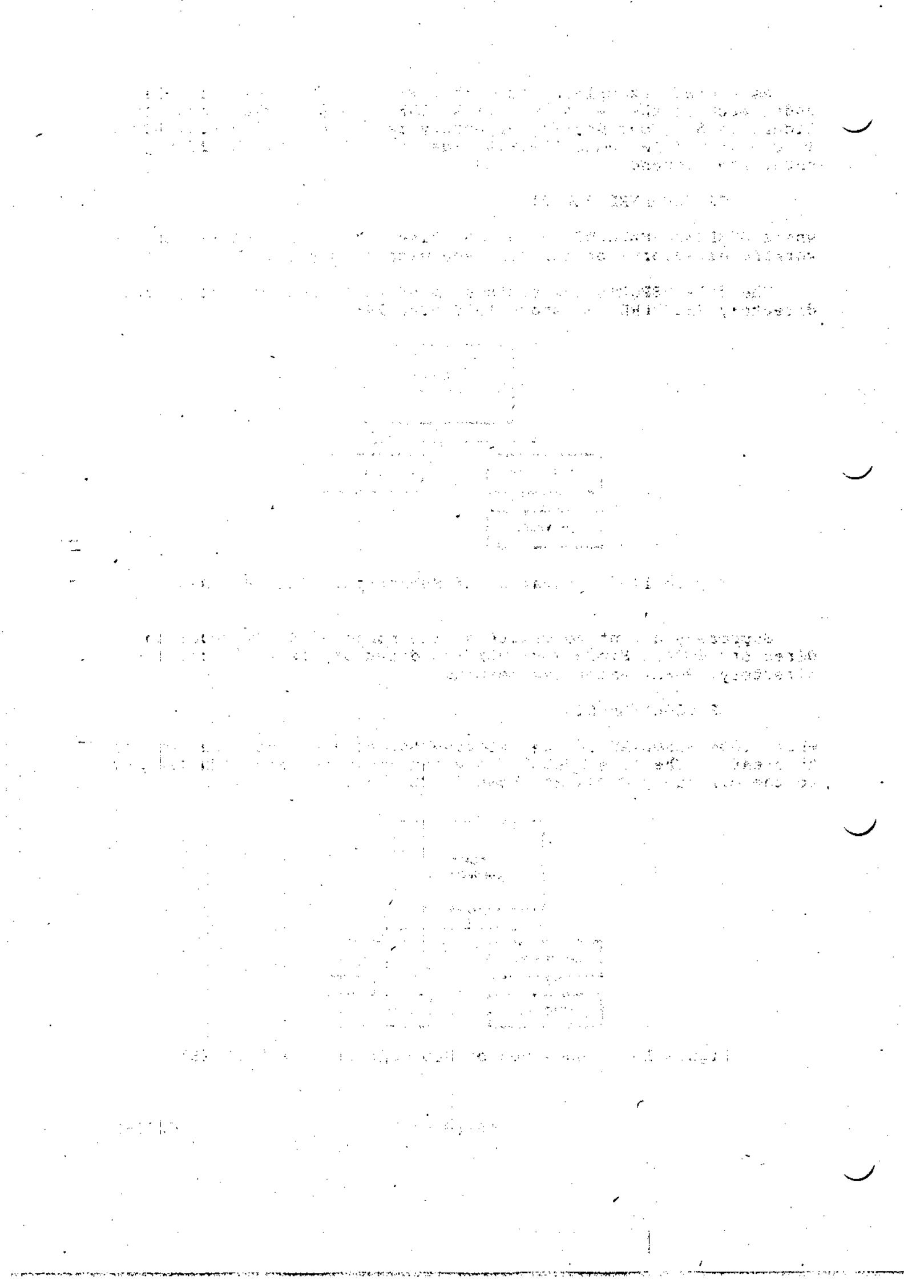


Figure 14-6 Location of Subordinate File WORDLIST



in the directory ^SYSVLA>UDD>PROGS>TOOLS, and wanted to copy in the file ^SYSVLA>UDD>PROGS>COOK>REC3.A, he needs to type only:

```
CP <COOK>REC3.A
```

The command copies REC3.A into TOOLS and names it REC3.A by default. You must be in the target directory to use this feature.

For another example, to copy cards onto a tape named BS001, that is already mounted, enter:

```
CP !CDR00 !MT900>BS001>WESTNAMES
```

LOCATING FILES

You can use the Where (WH) command to locate and display a file's full pathname. The system will search your working directory and the two system libraries, SYSLIB1 and SYSLIB2, looking for your file. If the file is found, its full pathname is displayed. If the file is not found, an error message is displayed. You may find this command useful if you know the simple pathname of a file but want to know its absolute pathname, or, to determine if the file you want to locate exists.

LISTING FILES AND DIRECTORIES

You can list the contents of any directory that you have at least list access to by using the List Names (LS) command.

For example, Cook lists the contents of his working directory by entering:

```
LS

DIRECTORY:  ^SYSVL1>UDD>PROGS>COOK

                        PHYSICAL    STARTING      RECORD
ENTRY NAME    TYPE      SECTORS     SECTOR HEX    LENGTH
  **************************************************
START_UP.EC    S           8           580          256
  **************************************************
```

The record length is the number of characters per line. Listing Cook's file with the -BF argument would produce this information.

DIRECTING OUTPUT TO A FILE

To direct output to a file (which need not have been previously created) using the FO (File Out) command, enter:

```
FO FILEA
```

All normal system output (such as a response to an LS command) will go to FILEA, which is your new user-out file. Error messages and the ready message that go to the error-out file cannot be redirected and will continue to appear at your terminal. Thus, if you entered an LS command, the system would write the listing to FILEA and respond at your terminal with only the ready message. However, input directed to your terminal is unaffected by the FO command.

DIRECTING OUTPUT TO A PRINTER

If you are performing functions that will lead to many pages of output, you can direct output to a printer. The command:

```
FO !LPT00
```

directs all subsequent output to LPT00 (assuming that you have access to the printer). Note that while you are using the printer, no one else can use it. In a multi-user system you may wish to avoid tying up the printer. (See "Deferred Printing" for information on printing large files.)

REDIRECTING OUTPUT TO YOUR TERMINAL

After you have finished directing output to a printer, you should redirect output to your terminal. Enter the FO command with no arguments:

```
FO
```

(The default is to redirect output to your terminal.)

Printing Control

You can print files at your terminal or you can request deferred printing. If you use the Print (PR) command, output appears on your terminal (i.e., output goes to the user-out file). This is inconvenient, however, if you are printing large files. For large files, you have the option of using deferred printing. The system will store your print request in a first-in, first-out queue.

NOTE

Deferred print requests are queued on disk and are not lost when the system is restarted.

Program Execution

Most of the programs you write will require some type of input and output. Before you execute a program, you must provide information that tells the program where your input will come from and where your output will go. The GET and REMOVE commands allow you to reserve files and devices for program input and output, and, after program execution, to cancel those reservations.

GET performs two functions. First, it reserves a file or device for use by the executing program. This reservation may set exclusive access or some degree of shared access (see "Reserving Files or Devices"). Secondly, GET establishes a relationship between pathnames and the logical file numbers (LFNs) by which you can gain access to files and devices.

Once program execution has terminated, you can use the REMOVE command to cancel file/device reservations and the LFNs that your program assigned with the GET command.

For example, you are compiling the Assembly Language program CARDIN. CARDIN uses two files, a card reader (from which input will be read) and a disk file (to which output will go). The program refers to these two files by logical file numbers (LFNs) 1 and 3.

After linking your object unit into a bound unit, you must use the GET command to reserve an input file (a card reader) and an output file (a disk file).

To reserve the card reader, specify:

```
GET !CDR00 -LFN 1
```

To reserve the disk file, specify:

```
GET AS_DIR>MASTER -LFN 3
```

In this example, it is assumed that the file MASTER was previously created under the directory AS_DIR. It is also assumed that the directory AS_DIR is subordinate to your working directory. (The GET command could have directed program output to any file, not necessarily one named MASTER.)

To send mail to another person, you might enter:

```
MAIL LOWELL
```

where LOWELL is the person_id of the receiver. The system will respond:

```
INPUT:
```

You can then enter the text of your message. Terminate the message by entering a period (.) or the letter Q followed by a carriage return. Your message is queued in Lowell's mailbox until Lowell issues a MAIL command to display mail.

To mail a file that might be a program or a long message for many users, use the filename argument of the MAIL command:

```
MAIL LOWELL HEX_AS.A
```

This command mails the file HEX_AS.A to Lowell. Long messages should not be sent to users at a VIP terminal.

NOTE

Before you can receive mail, either you or your system operator must have previously created the mailbox directory and the necessary mailboxes, and have set access controls on these mailboxes. See the System User's Guide manual for details.

ABSENTEE PROCESSING

MOD 400 offers both interactive and absentee (batch) processing. As an absentee user, you submit requests against the system batch task group ($B). Absentee processing allows you to perform multiprocessing on the system; i.e., you can process interactive tasks while the system processes one or more of your absentee requests simultaneously. All system software components are available to you as an absentee user.

The system operator creates the batch task group against which all users place requests on a queued first-in, first-out basis. To enter a request into the batch queue, use the Enter Batch Request (EBR) command. The EBR command requires you to specify a command-in file containing commands to be executed in the batch task group. Normally, you create this file on disk in your working directory before entering your batch request.

Section 15
LINE EDITOR

This section describes Line Editor functions and the Line Editor directive set.

OVERVIEW

The Line Editor creates and/or alters character text that constitutes files; the files usually are source unit files. The statements in a source unit file can be written in FORTRAN, COBOL, BASIC, PASCAL, or Assembly language. Throughout this section, it is assumed that source unit files are being edited.

Editing is controlled by directives entered to the Line Editor through the device specified in the in_path argument of the Enter Batch Request (EBR) or Enter Group Request (EGR) command. This device can be reassigned in the command that loads the Line Editor.

All editing is done in a temporary work area called the current buffer. When the Line Editor is invoked, the Line Editor creates a current buffer. To save Line Editor output, you must write the source unit contents of the current buffer to a file.

LINE EDITOR SUFFIX CONVENTIONS

When you create a source unit, you should append the appropriate suffix identification character to the name of the file that will contain the source unit. The suffix designates the type of text that constitutes the source unit. The suffix must be .C for COBOL programs, .F for FORTRAN programs, .PS for PASCAL programs, .B for BASIC programs, and .A for Assembly language programs.

When you specify the file names of Line Editor input and output files (in Line Editor directives), the editor requires that you designate the complete file name, including any of the suffixes just listed. The Line Editor does not append a suffix to its input and output files.

LINE EDITOR DIRECTIVE FORMAT CONVENTIONS

Most Line Editor directives consist of only a directive name, a directive name preceded by one or two addresses, or a directive name optionally preceded by one or two addresses and followed by text and termination escape characters (!F) that designate the end of the directive and cause the Line Editor to switch from input mode to edit mode. These formats are illustrated here. Note that if a directive includes text, the text may be specified beginning immediately after the directive name (see Format 4) or beginning on the next line (see Format 5).

FORMAT 1:

dirname

FORMAT 2:

adr_1 dirname

FORMAT 3:

$$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \text{ dirname}$$

FORMAT 4:

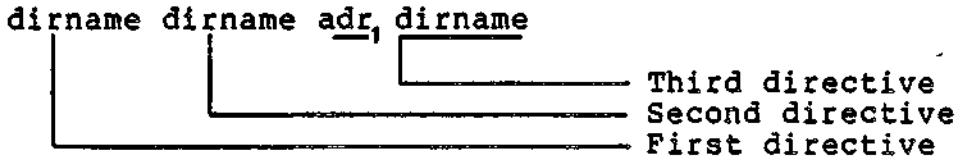
$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] \text{dirname[text]!F}$$

will be used. Address default values are described later in this section under each directive's argument descriptions.

Multiple Line Editor directives can be entered on a single line; it is not necessary to separate each directive with a delimiter, but one or more spaces can be specified, as illustrated below:

Directives not separated by delimiters:

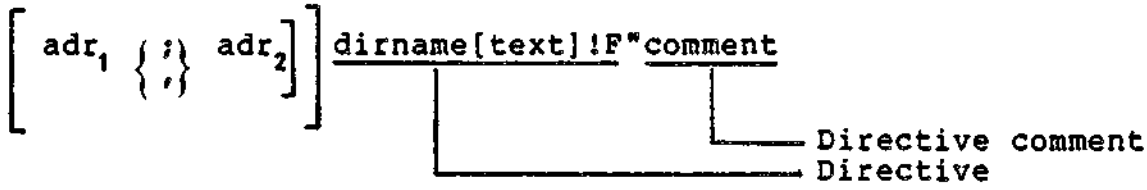
Directives separated by delimiters:



A comment can be included at the end of a directive line (i.e., at the end of the last or only directive); the comment must be preceded by a quotation mark ("), as illustrated:

```
adr  dirname dirname"comment
```

To include a comment after an input mode directive, specify the comment after the terminator !F; otherwise, the comment is included as text.



If a terminal is the directive input device, press RETURN at the end of each line.

Methods of Specifying Addresses

Each address can be specified by one of the following methods or by a combination of these methods:

- Number of line
- Position of line relative to the "current" line
- Contents of line.

You can locate lines relative to the current line by specifying an address that consists of a period followed by one or more signed decimal numbers. For example, the address .+1 specifies the line immediately following the current line, the address .-1 specifies the line immediately preceding the current line, and .+5+5-3 specifies the seventh line after the current line.

When specifying an increment to the current line number, you can omit the plus (+) sign; e.g., .5 is interpreted as .+5. When specifying a decrement to the current line number, you can omit the period; e.g., -3 is interpreted as .-3, and .5+5-3 is interpreted as .+7.

DESIGNATING CONTENTS OF LINE AS AN ADDRESS

You can designate that the Line Editor locate the first line that contains a specified character or a specified sequence of characters by designating those characters in an expression as an address. An expression comprises one or more ASCII characters, which must be delimited by slashes (e.g., /ASCII characters/).

The Line Editor will search the lines in the current buffer until it finds the first occurrence of the specified expression; unless specified otherwise,* the expression can be in any position within the line. The Line Editor searches from the line immediately following the current line (i.e., .+1) through the last line in the buffer; if a line containing the specified expression is not found, the Line Editor then searches line 1 to the current line. In the directive format:

```
/BBB/dirname
```

the address is the expression BBB. The Line Editor searches as many lines as necessary for the first occurrence of BBB. The contents of the source unit being searched are listed below. (The numbers within parentheses represent line numbers.)

```
(1) AAA
(2) BBB
(3) CCC (current line)
(4) BBB
```

The specified directive causes the Line Editor to locate line number 4, since this is the first line after the current line that contains the expression BBB.

---

*If a circumflex (^) is designated as the first character of the expression, the expression must be the first expression on the line; if $ is designated as the last character of the expression, the expression must be the last expression on the line. Use of these special characters is described in the following paragraphs.

3. For the Line Editor, two hexadecimal characters can be interpreted as one ASCII byte by using the escape sequence !Hxx, where xx are the two hex characters. However, this feature must be used with care since some of the hexadecimal characters may be confused with control or special characters in ASCII strings. The following is a list of the hexadecimal characters whose use is restricted:

   0A is the line feed character; in a string expression, it is interpreted as a request for advancement to a new line.

   2E and AE in a regular expression are treated as ".".

   26 and A6 in a string expression are treated as "&".

   2A and AA in a regular expression are treated as "*".

   24 at the end of a regular expression is interpreted as "end-of-line ($)".

   5E at the beginning of a regular expression becomes "beginning-of-line (^)".

   Rather than attempting to substitute in an expression using the characters above, execute a Change directive, reentering the line using hexadecimal and ASCII characters for the entire line.

Following are some examples of expressions specified as addresses in Line Editor directives. Following each expression is a description of the line/character(s) in the current buffer for which the Line Editor will search. In each case, the Line Editor searches the lines sequentially, starting with the line immediately following the current line to the end of the file, and then from line one through the current line.

Expression	Description
/A/	Locates the first line that contains the expression A in any position in that line.
/ABC/	Locates the first line that contains the expression ABC in any position on that line.
/AB*C/	Locates the first line that contains the expression AC or A followed by any number of Bs and a C.

COMPOUND ADDRESSES

An address can be formed by combining any of these methods. If a compound address contains a line number, the line number must be the first element of the address.

The first element of the compound address determines the starting location from which the Line Editor will search for the designated expression. If the first element is a line number, the Line Editor searches for the expression starting with the line that immediately follows the specified line number. (Ordinarily, the Line Editor searches starting with the line that immediately follows the current line.)

Example 1:

```
10/ABC/
```

The Line Editor searches the lines in the current buffer for the characters ABC, starting with line 11.

Example 2:

```
.-8/ABC/
```

The Line Editor searches the lines in the current buffer for the characters ABC, starting eight lines before the current line.

Example 3:

```
/ABC//DEF/
```

The Line Editor searches for the first line containing DEF that occurs after the first line containing ABC.

Each expression in a compound address can be followed by a signed decimal integer.

Example 4:

```
/ABC/-10/DEF/5
```

The Line Editor searches for the first occurrence of the character string DEF that is within 10 lines before the first line that contains ABC. After DEF is found, the current line is the fifth line after the line containing the match for DEF.

```
(1) ABC
(2) DEF (current line)
(3) GHI
(4) ABC
(5) XYZ
(6) ABC
```

These addresses specify the line immediately following the current line through the second line after the current line. The Line Editor locates lines 3 and 4. Line 4 becomes the current line.

Example 5:

```
.1; 2dirname
```

These addresses are the same as those in Example 4, but they are separated by a semicolon. If the contents of the sample source unit are the same as in Example 4, this directive causes the Line Editor to locate lines 3, 4, and 5. This first address specifies the line immediately after the current line, i.e., line 3. Line 3 then becomes the current line. The second address specifies that the Line Editor locate through the second line after the (new) current line, i.e., lines 4 and 5.

As the next example illustrates, the same series of lines can be requested by specifying their addresses in more than one way, using different delimiters.

Example 6:

```
/ABC/,/ABC/+3dirname
/ABC/;.+3dirname
```

The contents of a sample source unit follows. The numbers within parentheses represent line numbers.

```
(1) ABC
(2) DDD (current line)
(3) EEE
(4) FFF
(5) GGG
(6) HHH
```

The first series of addresses specifies that the Line Editor locate the first line that contains ABC (line 1) through the third line after that line (lines 2, 3, and 4). Line 4 becomes the current line.

The second series of addresses specifies that the Line Editor locate the first line that contains ABC (line 1), make that line the current line, and then reference three lines from the "new" current line (lines 2, 3, and 4). Line 4 becomes the current line.

{-NO_BLANK_SUPPRESS}
{-NBS}

No blank suppression; i.e., the Line Editor does not suppress trailing blanks on the input line (for one invocation only). Subsequent invocations without -NBS will suppress trailing blanks.

{-FILE_SIZE nn}
{-FS nn}

Alter the initial size of the work file to the size in the user-supplied value of nn, where nn is a decimal integer of up to four characters and designates the number of 256-byte control intervals. If an output file is created, it is initialized to the same size.

Default: 4.

{-ARGS strings}
{-ARG strings}

Up to nine character strings that are numbered sequentially and may be passed to the Line Editor in the "Change Origin of Text During Edit Mode" (!B) Line Editor directive. Each argument following the -ARG keyword is copied to buffer (ARGn). n denotes the position of the argument following the -ARG and can be any value from one through nine. If specified, this argument and its strings must be entered last.

{-SAFE name}
{-SF name}

Permanent work files called name.EDWK1 and name.EDWK2 contain the latest copy of the current buffer. Name can be from one to six characters. Abnormal termination causes the work files to be closed in their current state and saved for later use, and normal termination releases them. To reuse the work files, invoke the Line Editor without -SAFE or with -SAFE and a different name.

Default: Work files are temporary files and are released under all conditions.

Table 15-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive/ Escape Sequence	Function	Topic Under Which Described
E	Execute command other than Line Editor without exiting from the Line Editor.	Execute directive (advanced usage -- general)
G	Search for specified line(s) that contain specified character string.	Global directive (advanced usage -- general)
I	Add line(s) before a specified address.	Insert directive (input mode).
K	Copy line(s) in current buffer to specified auxiliary buffer. Do not delete lines from current buffer. Overlay existing line(s) in auxiliary buffer.	Copy directive (advanced usage -- auxiliary buffers)
L	Send line feed to the user-out file.	Line Feed directive (advanced usage -- general)
M	Move line(s) from current buffer to specified auxiliary buffer; delete the lines from current buffer and overlay existing line(s) in auxiliary buffer.	Move directive (advanced usage -- auxiliary buffers)
N	Designate different line as the current line.	New Current Line directive (advanced usage -- general)
P	Print specified line(s) in current buffer.	Print directive (edit mode)
Q	Conditionally terminate execution of Line Editor.	Quit directive (edit mode)
R	Read text from file to current buffer.	Read directive (edit mode)

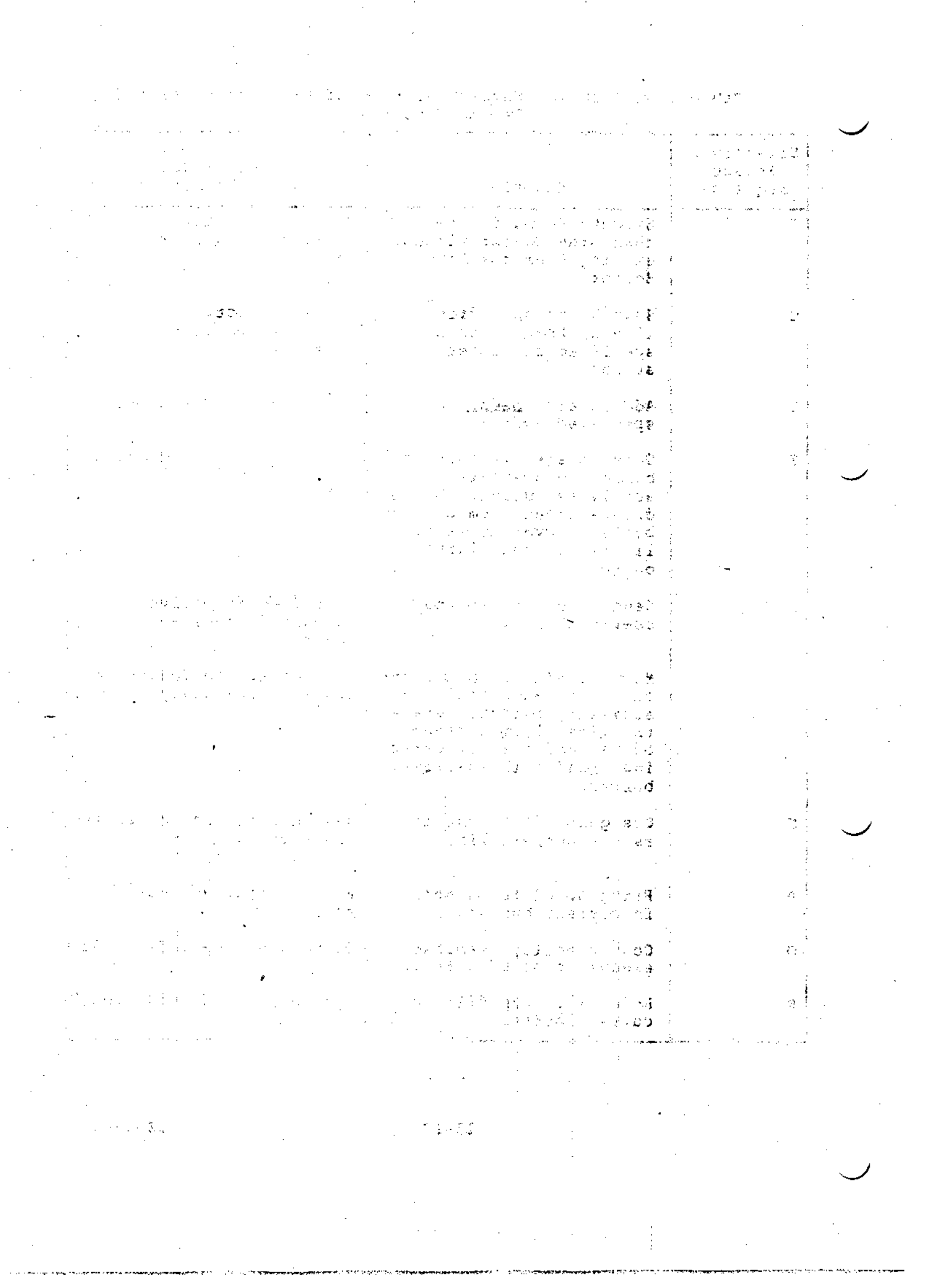


Table 15-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive/ Escape Sequence	Function	Topic Under Which Described
!Hxx	Interpret two following hexadecimal characters as one ASCII byte.	
!K	Copy line(s) in current buffer to specified auxiliary buffer; do not delete existing line(s) in auxiliary buffer.	Copy-append directive (advanced usage -- auxiliary buffers)
!L	Send line feed to the error-out file.	Line feed directive (advanced usage -- general)
!M	Move line(s) from current buffer to specified auxiliary buffer; delete the line(s) from current buffer and append them to existing line(s) in auxiliary buffer.	Move-append directive (advanced usage -- auxiliary buffers)
!P	Type line number and contents of specified line(s) in current buffer.	Print With Line Number directive (advanced usage -- general)
!Q	Unconditionally terminate execution of Line Editor.	Quit directive (edit mode)
!R	Accept single line from terminal.	Accept Single Line from Terminal directive (advanced usage -- auxiliary buffers)
!T	Display line of text on user-out file; subsequent input/output will be on the same line.	Type directive (advanced usage -- programming)
!U	Convert specified lowercase expression to uppercase.	Uppercase directive (advanced usage -- general)
!?	Cause message indicating whether input or edit mode is in effect.	

Table 15-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive/ Escape Sequence	Function	Topic Under Which Described
:	Define location to which Line Editor can be directed for subsequent directive(s).	Label directive (advanced usage -- programming)
=	Type line number of specified line in current buffer.	Print Line Number directive (advanced usage -- general)
>	Accept subsequent directive(s) from specified location in current buffer or interactively.	Go To directive (advanced usage -- programming)
?	If specified line is in current buffer, execute specified directive(s).	Address Prefix directive (advanced usage -- programming)
"	Annotate Line Editor files.	Comment directive (advanced usage -- programming)

CREATING A SOURCE UNIT

To create a source unit, perform the following steps.

1. Change the working directory to a user volume by specifying the Change Working Directory (CWD) command (see the Commands manual).

2. Load the Line Editor. (See "Loading the Line Editor" earlier in this section.)

3. If there already are lines in the current buffer, clear the buffer by entering: 1,$D.

4. Enter the appropriate Input directive and text to be included.

5. Make changes, if necessary, by entering the appropriate Input and/or Edit directive(s).

6. Write the contents of the current buffer to a file by using the Write directive.

You can create a source unit by using the Append or Insert directive. You can add lines to an existing source unit by using any or all of the above directives.

Each input directive must have one of the following formats:

FORMAT 1:

$$\left[\ adr_1\left[\left\{\begin{matrix};\\,\end{matrix}\right\}adr_2\quad dirname\right]\right]$$

```
[text]
  .
  .
  .
!F["comment]
```

FORMAT 2:

$$\left[\ adr_1\left[\left\{\begin{matrix};\\,\end{matrix}\right\}adr_2\right]\right]dirname[text]!F["comment]$$

If directives are being entered through a terminal, the directive can either be immediately followed by a carriage return and then text (i.e., the lines to be included in the source unit), or the directive name can be immediately followed by text, with additional lines of text (if any) added on subsequent lines. The text can be any number of lines of ASCII characters. The maximum number of characters per line is determined by the value specified in the -LINE_LEN n argument of the ED command. The last line of text must be followed by the escape sequence !F to terminate input mode; otherwise, the next Line Editor directive is interpreted as additional text. The escape sequence !F can be entered at the end of the last line of text or in the first character position of the next line. The next directive can begin in the next character position or on the next line.

NOTES

1. To enter a blank from the operator terminal, as the first character on a line, precede it with an !C sequence.

2. The characters !F can be included as text by preceding them with !C; in this case, !F does not designate the end of the text.

3. When entering directives from a card reader, the punch for an exclamation point is 12-8-7.

Input directives are described in detail on the following pages. In the examples, numbers in parentheses are references to line numbers and do not appear in memory or in text.

Example 1, Creating a New Source Unit:

In this example, the buffer is empty.

```
A
WWW
XXX
YYY
ZZZ
!F
```

This Append directive puts lines WWW, XXX, YYY, and ZZZ into the current buffer. Since the buffer is empty, it is not necessary to specify an address. The lines will be inserted, in the order in which they were entered, starting at line 1. The lines put into the buffer constitute a new source unit which can then be edited and/or written to a file.

Example 2, Adding Lines to an Existing Source Unit:

```
/TTT/A
UUU
!F
3A
WWW
XXX
!F
```

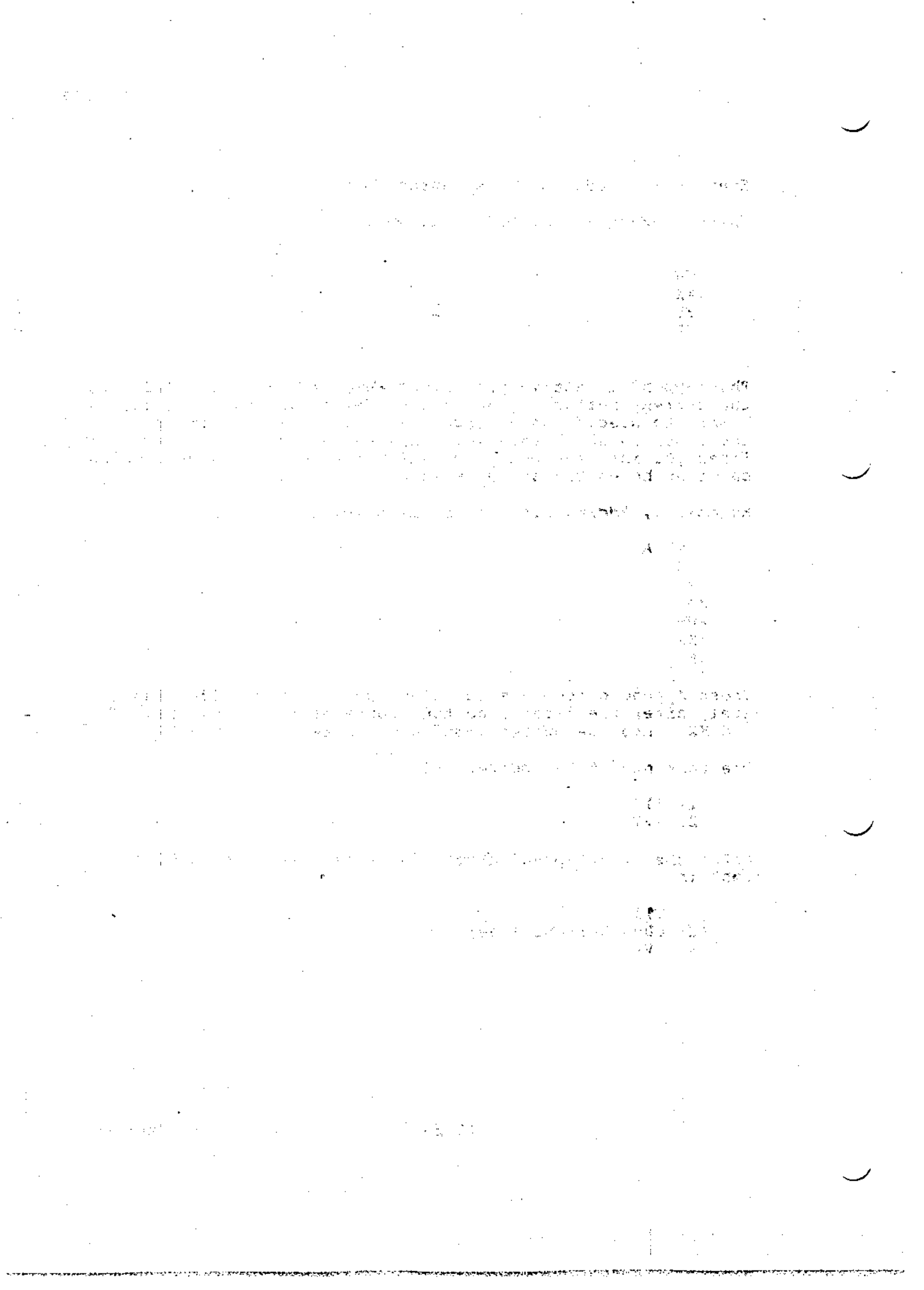
These Append directives put line UUU into the buffer immediately after the first line that contains TTT, and lines WWW and XXX into the buffer immediately after the third line.

The contents of the buffer are:

```
(1) TTT
(2) VVV
```

After the first Append directive is executed, the buffer will contain:

```
(1) TTT
(2) UUU (current line)
(3) VVV
```



CHANGE (C)

Delete a single line or a series of lines in the current buffer and then insert the text specified between the directive name and the insert terminator !F.

After the Change directive is executed, the current line is the last line of inserted text. The inserted line(s) are given line numbers and subsequent lines, if any, are renumbered.

FORMAT 1:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] C$$

text
.
.
.
!F

FORMAT 2:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] Ctext!F$$

ARGUMENTS:

adr_1

Address of the first or only line to be deleted and replaced.

Default: Current line.

adr_2

Address of the last line to be deleted and replaced.

Default: Only the line identified by adr_1 is deleted and changed.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is deleted and replaced.

Example 3:

```
.,5C        .,$C
XXX    or   XXX
!F          !F
```

Each of the Change directives above deletes the current line through line 5 and replaces them with a single line containing XXX.

After the change directive is executed, the buffer will contain:

(1) AAA
(2) BBB
(3) XXX (current line)

Example 1:

In this example, the current buffer is empty.

```
I
AAA
BBB
CCC
DDD
!F
```

This Insert directive creates in the current buffer a new source unit comprising lines AAA, BBB, CCC, and DDD. The lines can then be edited and/or written to a file.

In Examples 2, 3, and 4, the contents of the current buffer are:

```
(1) AAA
(2) BBB
(3) CCC
(4) DDD (current line)
```

Example 2:

```
-2I
XXX
!F
```

This Insert directive designates that a line containing XXX be inserted two lines before the current line.

After the Insert directive is executed, the current buffer will contain:

```
(1) AAA
(2) XXX (current line)
(3) BBB
(4) CCC
(5) DDD
```

Example 3:

```
/AAA/I
H!C!FH
KKK
!F
```

EDIT MODE DESCRIPTION AND DIRECTIVES

In Edit mode, you can write to file text originated in input mode, call up existing files, edit them, print them, and exit from the Line Editor.

Edit mode directives have the following capabilities:

- Delete specified line(s) from the current buffer (Delete directive)
- Print on the user-out file specified line(s) in the current buffer (Print directive)
- Terminate execution of the Line Editor (Quit directive)
- Read text from specified file into the current buffer (Read directive)
- Substitute a designated string of characters in specified line(s) with another specified string of characters (Substitute directive)
- Write specified line(s) from the current buffer to specified file (Write directive).

NOTES

1. To edit an existing source unit, the Read directive must be previously specified.
2. Until you are familiar with the Line Editor, enter Print directives frequently so you can determine the status of the lines being edited.
3. To save the results of an edited or newly created source unit, you must specify the Write directive before you terminate execution of the Line Editor.

Most edit mode directives have one of the following formats:

FORMAT 1:

```
dirname["comment]
```

FORMAT 2:

```
adr1 dirname["comment]
```

FORMAT 3:

$$\left[adr_1 \left\{\begin{matrix};\\,\end{matrix}\right\}\right] adr_2\ dirname["comment]$$

DELETE (D)

Delete a single line or consecutive lines from the current buffer.

After the Delete directive is executed, each subsequent line in the buffer is renumbered, and the current line is the line that immediately follows the last line deleted or the last line in the buffer if the previous "last line" was deleted.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2 \right] \right] D$$

ARGUMENTS:

adr_1

Address of the first or only line to be deleted.

Default: Current line.

adr_2

Address of the last line to be deleted.

Default: Only the line identified by adr_2 is deleted.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is deleted.

In the following examples, the contents of the current buffer are:

(1) AAA
(2) BBB (current line)
(3) CCC
(4) DDD
(5) EEE

PRINT (P)

Print a single line or consecutive lines in the current buffer. You can specify the address(es) of the line(s) to be printed, or you can request a printout of the first line that contains a specified expression. The printout is issued to the user-out file; i.e., the file designated in the -OUT out_path argument of the Enter Batch Request (EBR) or Enter Group Request (EGR) command, unless the file was reassigned in the File Out (FO) command. If the printout occurs on the operator terminal, each line of text is preceded by the group identification characters.

After the Print directive is executed, the current line is the last (or only) line printed.

FORMAT 1:

Format including directive name P:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] P$$

ARGUMENTS:

adr_1

Address of the first or only line to be printed. The Line Editor begins its search at the second line in the current buffer.

Default: Current line.

adr_2

Address of the last line to be printed.

Default: Only the line identified by adr_1 is printed.

NOTE

If both adr_1 and adr_2 are omitted and P is specified, only the current line is printed.

FORMAT 2:

Format excluding directive name P:

$$adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right]$$

Example 3:

```
4P
```

This Print directive causes a printout of line number 4.

```
GGGHHH
```

After this directive is executed, the current line is line number 4.

Example 4:

```
.,4P
```

This Print directive causes a printout of the current line (line number 2) through line number 4:

```
CCCDDD
EEEFFF
GGGHHH
```

After this directive is executed, the current line is line number 4.

Example 5:

```
/AAA/
```

This Print directive causes a printout of the first line that contains AAA.

```
AAABBB
```

After this directive is executed, the current line is line number 1.

Example 6:

```
3D/AAA/
```

This directive line contains (1) a Delete directive and (2) a Print directive in which only an expression is designated.

This directive line deletes line number 3 and causes a printout of the first line that contains AAA. After the directives are executed, the current buffer will contain:

```
(1) AAABBB
(2) CCCDDD
(3) GGGHHH
```

QUIT (Q OR !Q)

Exit from the Line Editor. Quit must be specified at the end of the editing session. This directive must be the last or only directive on a line. If the directive input device is a terminal, the Quit directive must be immediately followed by a carriage return.

Quit is executed conditionally or unconditionally, depending on which Quit format is specified. In a conditional Quit request (Format 1), if a buffer has a pathname associated with it via a Read or Write directive and the contents of the buffer have been modified but not written to a file before the Quit directive is entered, a warning message is issued and Quit is not executed. After the message, any Line Editor directive(s), including Write, may be entered. If Write is not specified and Quit is reentered, the Quit directive is executed and changes specified in previous Line Editor directives are not saved. In an unconditional Quit request (Format 2), modified buffers are not checked before Quit is executed.

FORMAT 1:

```
Q
```

FORMAT 2:

```
!Q
```

Example:

A	Append directive, which puts specified lines into current buffer.
AAABBB CCCDDD EEEFFF	Lines that will be put into current buffer.
!F	Designate the end of the insertion.
2D	Delete the second line of text (e.g., CCCDDD).
W FIRST	Write all lines in buffer to file named FIRST.
Q	Return control from the Line Editor to the Command Processor.

READ (R)

Read text from a specified file into the current buffer. The Read directive must be the only or last directive on a line. After the Read directive is executed, the current line is the last line read from the file.

FORMAT:

[adr]R[path]

ARGUMENTS:

adr

Address of a line in the current buffer; the contents of the specified file will be appended after this line.

Default: Last line in the buffer; if the buffer is empty, the file is appended starting at the first line in the buffer.

path

Pathname of the ASCII file to be read into the current buffer. (Methods of specifying pathnames are described in Section 14.) The pathname may be preceded by any number of blanks.

Default: Pathname specified in the latest Read or Write directive associated with the current buffer. To determine which pathname was specified last, specify the Buffer Status directive, which is described under "Advanced Usage of the Line Editor" later in this section. If the path argument is not specified and a pathname was not previously specified, an error message is issued.

NOTE

!CDR or any other device name beginning with an exclamation point (!) may cause errors. The exclamation point is a Line Editor escape character. A read of !CDRxx (R !CDRxx) will try to read file name DRxx because !C is a conceal flag. Use >SPD> in place of the exclamation point (e.g., R >SPD>CDRxx), or conceal a C (e.g., R !!CCDRxx).

Example 1:

```
R START
```

This Read directive reads into the current buffer the contents of a file whose simple pathname is START. Since an address is not specified, the lines are read into the buffer after the last line that currently is in the buffer.

The contents of START are:

```
(1) AAA
(2) BBB
(3) CCC
```

If the buffer is empty, after the Read directive is executed, the current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC (current line)
```

If the buffer already contains:

```
(1) XXX
(2) YYY
(3) ZZZ
```

After the Read directive is executed, the current buffer will contain:

```
(1) XXX
(2) YYY
(3) ZZZ
(4) AAA
(5) BBB
(6) CCC (current line)
```

Example 2:

```
/CCC/R NEW
```

This directive reads the contents of the file whose simple pathname is NEW into the current buffer after the first line in the current buffer that contains CCC.

The contents of the current buffer are:

```
(1) AAA
(2) BBB (current line)
(3) CCC
(4) CCC
```

The contents of NEW are:

```
(1) XXX
(2) ZZZ
```

After the Read directive is executed, the current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC
(4) XXX
(5) ZZZ (current line)
(6) CCC
```

Example 3:

This example illustrates the Read directive used in conjunction with Append and Write directives. The current buffer is empty.

```
A         Puts subsequent lines into the current buffer.
AAA
BBB
CCC
.F        Designates the end of the insert.
W  NOW    Writes the contents of the current buffer to the
          file whose simple pathname is NOW.
R         Reads into the current buffer, after the last line
          in the buffer, the contents of NOW; NOW is the
          pathname specified in the last Write directive.
```

After the Read directive is executed, the current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC
(4) AAA
(5) BBB
(6) CCC (current line)
```

SUBSTITUTE (S OR !S)

Replace each occurrence of a specified string of characters in a single line or in a sequence of lines with another specified string of characters.

After this directive is executed, the current line is the last line located by the Line Editor.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix};\\,\end{Bmatrix} adr_2\right]\right] S/regexp/string/$$

or

$$\left[adr_1 \left[\begin{Bmatrix};\\,\end{Bmatrix} adr_2\right]\right] !S/regexp/string/$$ (See Note 3)

ARGUMENTS:

adr_1

Address of the first line to be searched for the specified string of characters. The search begins at the second line in the current buffer.

Default: Current line.

adr_2

Address of the last line to be searched for the specified string of characters.

Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is searched.

/

(Delimiter) Can be any character that is not in regexp or string. However, the same delimiter must be used in each of the three locations where a delimiter is required.

regexp

String of characters for which the Line Editor is searching; each occurrence of this character string within the specified addresses will be replaced with the character(s) specified in the argument "string".

Default: The last regexp specified. This can be determined by entering the ZREGEXP directive, which is described under "Line Editor Debugging Directives".

string

String of characters that will replace each occurrence of regexp.

NOTES

1. If string contains the character "&" in any position, each occurrence of regexp to be replaced will be replaced with regexp included in string, in place of "&". For example, if regexp is "in" and string is "&to", each occurrence of "in" becomes "into". To cancel the special meaning of "&", precede it with !C.

2. The occurrence of a line feed in the string expression determines the new line characters, i.e., point in the resulting line at which the line is to be split into two lines.

3. If the directive name !S is used (as illustrated in the second directive format) and the specified substitution fails, no error message is issued and execution of the command file (if any) continues.

Example 1:

```
S/ABGDEF/ABC linefeed DEF/
```

This Substitute directive searches the current line and (1) replaces each occurrence of ABGDEF with ABCDEF and (2) causes the character string to be split between two lines. ABC will be on the first line, and DEF will be on the second line.

Example 2:

The contents of the current buffer are:

```
(1) E
(2) NTE
(3) R
(4) YOUR
```

```
1,3S/linefeed key//
```

After this Substitute directive is entered, the current buffer will contain:

```
(1) ENTERYOUR
```

In the following examples, the contents of the current buffer are:

```
(1) AAACCC
(2) BBBAAA (current line)
(3) CCCBBB
(4) DDDAAA
```

Example 3:

```
2,4S/AAA/XXX/
```

This Substitute directive searches lines 2 through 4 and replaces each occurrence of AAA with XXX.

After this directive is executed, the current buffer will contain:

```
(1) AAACCC
(2) BBBXXX
(3) CCCBBB
(4) DDDXXX (current line)
```

Example 4:

```
.,4S-CCC-UUU-
```

This Subsitute directive searches the current line (line 2) through line number 4 and replaces each occurrence of CCC with UUU.

After this directive is executed, the current buffer will contain:

(1) AAACCC
(2) BBBAAA
(3) UUUBBB
(4) DDDAAA (current line)

Example 5:

```
-1,/DDD/S//&JJJ/
```

This Substitute directive searches one line before the current line (line 1) through the first line that contains DDD (line 4) and replaces each occurrence of DDD with DDDJJJ.

After this directive is executed, the current buffer will contain:

(1) AAACCC
(2) BBBAAA
(3) CCCBBB
(4) DDDJJJAAA (current line)

Example 6:

```
/BBB/S//XXX/
```

This Substitute directive searches the first line after the current line through the current line (line 2) and changes the first occurrence of BBB to XXX.

After this directive is executed, the current buffer will contain:

(1) AAACCC
(2) BBBAAA
(3) CCCXXX (current line)
(4) DDDAAA

WRITE (W)

Write a specified line or a series of lines in the current buffer to a specified file. If the file does not already exist, a new file is created with the specified file name. If the named file does exist and currently contains other data, the line(s) written to the file via the Write directive replace the existing contents.

To save the results of previously specified Line Editor directives, you must specify the Write directive before you terminate execution of the Line Editor (i.e., Write must be specified before Quit).

The Write directive must be the last directive on a line. After the Write directive is executed, the specified line(s) remain in the current buffer; a copy of them is written to the specified file.

FORMAT:

$$\left[\ adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] W\ [path]$$

ARGUMENTS:

adr_1

Address of the first line to be written to a specified file.

Default: First line in the current buffer.

adr_2

Address of the last line to be written to a specified file.

Default: Last line in the current buffer.

NOTE

If both adr_1 and adr_2 are omitted, all lines in the current buffer are written to the specified file.

path

Pathname of the file to which the specified line(s) will be written. (Methods of specifying pathnames are described in Section 14.) The pathname may be preceded by any number of spaces. Default: Pathname specified in the latest Read or Write directive associated with the current buffer. If a pathname was not previously specified, an error message is issued.

Example 1:

```
W IDENT
```

This Write directive writes all lines in the current buffer to a file whose simple pathname is IDENT.

Example 2:

This example illustrates use of a Write directive in a sample Line Editor session. In this example, there is a file named EXIST that contains the following lines:

```
(1) AAA
(2) BBB
(3) CCC
(4) DDD
```

```
R EXIST
```

Read into the current buffer the contents of the file named EXIST. The current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC
(4) DDD (current line)
```

```
1,$S/AAA/XXX/
```

Search each line in the current buffer and change each occurrence of AAA to XXX. The buffer will contain:

```
(1) XXX
(2) BBB
(3) CCC
(4) DDD (current line)
```

```
1,3W
```

Write lines 1 through 3 to the file specified in the last Read or Write directive (i.e., EXIST). EXIST will contain:

```
(1) XXX
(2) BBB
(3) CCC
```

```
Q
```

Terminate execution of the Line Editor.

ADVANCED FUNCTIONS OF THE LINE EDITOR

The directives described on the previous pages permit you to create a source unit and perform basic editing. The following subsections describe Line Editor directives that perform general advanced functions, permit usage of auxiliary buffers, perform debugging, and perform programming functions. Within each subsection the directives are summarized and then described in detail alphabetically by full directive name.

GENERAL ADVANCED LINE EDITOR DIRECTIVES

The general advanced Line Editor directives have the following capabilities:

- Cause another specified directive to act on only those lines that do not contain a specified character string (Exclude directive)
- Permit execution of a command instead of Line Editor directives without exiting from the Line Editor (Execute directive)
- Cause another specified directive to act on only those lines that contain a specified character string (Global directive)
- Send line feed to user-out file and error-out file (Line Feed directive)
- Convert the specified expression to lowercase (Lowercase directive)
- Make a different line the current line (New Current Line directive)
- Print the line number of a specified line in the current buffer (Print Line Number directive)
- Print the line number and contents of specified line(s) in the current buffer (Print With Line Number directive)
- Convert the specified expression to uppercase (Uppercase directive).

EXCLUDE (V)

Exclude specified elements. The Exclude directive can be used in conjunction with Delete, Print, Print Line Number, and Print With Line Number directives so that the specified directive acts on only those lines that do not contain a specified character string.

After the Exclude directive is executed, the current line is the last line searched by the Line Editor (i.e., the line specified in adr_2 (see below)).

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] Vx/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched.

Default: First line in the current buffer.

adr_2

Address of the last line to be searched.

Default: Last line in the current buffer.

NOTE

If both adr_1 and adr_2 are omitted, all lines in the buffer are searched.

x

Directive name with which the Exclude directive is being isued; must be one of the following:

D - VD deletes line(s) that do not contain regexp.

P - VP prints the contents of line(s) that do not contain regexp.

!P - V!P prints the line number(s) and contents of line(s) that do not contain regexp.

= - V= prints the line number(s) of line(s) that do not contain regexp.

/

(Delimiter) Can be any character that does not occur in regexp. The same delimiter must be used before and after regexp.

regexp

String of characters for which the Line Editor will search; only lines that do not contain regexp will be acted upon by the Line Editor during execution of the directive name specified in argument x.

In the following examples, the contents of the current buffer are:

```
(1) JJJKKK (current line)
(2) LLLMMM
(3) NNNPPP
(4) RRRJJJ
```

Example 1:

```
1,3V!P/JJJ/
```

This Exclude Print with line number directive causes the Line Editor to search lines 1 through 3 and to print the line number and contents of each line that does not contain JJJ.

Typeout:

```
2 LLLMMM
3 NNNPPP
```

Current line: 3

Example 2:

```
VD*JJJ*
```

This Exclude Delete directive deletes each line that does not contain JJJ; since no addresses are specified, each line in the current buffer is searched.

After this directive is executed, the current buffer will contain:

```
(1) JJJKKK
(2) RRRJJJ (current line)
```

EXECUTE (E)

Cause execution processing. The Execute directive permits you to execute a command instead of Line Editor directives without exiting from the Line Editor; i.e., you can enter any command and then continue to use the Line Editor. For example, the Execute directive can be used to designate a printer as the Line Editor output file. Otherwise, if you want a printout of Line Editor output, the printout is issued to the terminal, which is the original user-out file. If the user-out file is a line printer and a Quit directive is entered to exit from the Line Editor, the user-out file remains set to the printer.

The Execute directive must be the last directive on a line.

The current line is not affected by Execute directives.

FORMAT:

```
E command
```

ARGUMENT:

command

Any command (see the Commands manual).

Example:

```
E FO  >SPD>LPT00
```

This Execute directive includes a File Out (FO) command, which sets the user-out file to the line printer whose pathname is >SPD>LPT00.

GLOBAL (G)

Act on only those lines that contain a specified character string and can be used in conjunction with Delete, Print, Print Line Number, and Print With Line Number directives.

After the Global directive is executed, the current line is the last line searched by the Line Editor.

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] Gx/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched.

Default: First line in the current buffer.

adr_2

Address of the last line to be searched.

Default: Last line in the current buffer.

NOTE

If both adr_1 and adr_2 are omitted, all lines in the current buffer are searched.

x

Directive name with which the Global is being used; must be one of the following:

D - Delete all line(s) in the specified range containing regexp.

P - Print the contents of line(s) containing regexp.

!P - Print the line number(s) and contents of line(s) containing regexp (see "Print With Line Number Directive" later in this section).

= - Print the line number(s) of line(s) containing regexp (see "Print Line Number Directive" later in this section).

/

(Delimiter) Can be any character that does not occur in regexp. The same delimiter must be used before and after regexp.

regexp

String of characters for which the Line Editor will search; only lines that contain regexp will be acted upon by the directive name specified in argument x.

In the following examples, the contents of the current buffer are:

(1) JJJKKK
(2) LLLMMM
(3) NNNPPP
(4) RRRJJJ

Example 1:

```
1,3G!P/JJJ/
```

This Global Print With Line Number directive causes the Line Editor to search lines 1 through 3 and print the line number and contents of each line that contains JJJ.

Typeout:

```
1 JJJKKK
```

Current line: 3

Example 2:

```
GD*JJJ*
```

This Global Delete directive deletes each line that contains JJJ; since no addresses are specified, all lines in the buffer are searched.

After this directive is executed, the current buffer will contain:

(1) LLLMMM
(2) NNNPPP (current line)

LINE FEED (L OR !L)

Send a line feed to the user-out file (L) or to the error-out file (!L). After the Line Feed directive is executed, the current line is unchanged. Default: none (addresses are ignored).

FORMAT:

L or !L

LOWERCASE (U)

Convert all occurrences of a specified expression within specified addresses from uppercase to lowercase. After the Lowercase directive is executed, the current line is the last line read.

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] U/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched.

Default: Current line.

adr_2

Address of the last line to be searched.

Default: adr_1.

regexp

String of characters for which the Line Editor searches. Only uppercase letters (A through Z) are converted; others are not changed.

Example:

```
U/ADR/
```

This Lowercase directive searches the current line and changes each occurrence of ADR to adr. The current line is:

```
ADR FIRST
```

After the Lowercase directive is executed, the line contains:

```
adr FIRST
```

NEW CURRENT LINE (N)

Cause the specified line to become the new current line. The contents of the new current line are not printed after the directive is executed.

FORMAT:

```
adrN
```

ARGUMENT:

adr

Address of the line that is to be the new current line.

Example:

```
/CCC/N
```

The following condition exists prior to execution of the N directive:

```
AAA (current line)
BBB
CCC
DDD
```

The situation will be as follows after the N directive is executed.

```
AAA
BBB
CCC (current line)
DDD
```

PRINT LINE NUMBER (=/!P)

Print out the line number of a specified line in the current buffer.

The printout is issued to the user-out file, i.e., the file designated in the -OUT out_path argument of the Enter Batch Request (EBR) or Enter Group Request (EGR) command, unless that file was reassigned.

After this directive is executed, the current line is the line whose line number was typed.

FORMAT:

[adr]=

ARGUMENT:

adr

Address of the line whose line number is to be typed.

Default: Current line.

In the following examples the contents of the current buffer are:

(1) AAABBB (current line)
(2) CCCDDD
(3) CCCEEE

Example 1:

/CCC/=

This Print Line Number directive causes a printout of the line number of the first line that contains CCC.

Printout:

2

Current line: 2

Example 2:

```
=
```

This Print Line Number directive causes a printout of the line number of the current line.

Printout:

```
1
```

Current line: 1

PRINT WITH LINE NUMBER (!P)

Print out the line number and contents of a single line or consecutive lines in the current buffer. The printout is issued to the user-out file, i.e., the file designated in the -OUT out_path argument of the Enter Batch Request or Enter Group Request command, unless the file was reassigned. If the printout occurs on a terminal, each line of text is preceded by the group identification characters.

After this directive is executed, the current line is the last line whose line number and contents were typed.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2 \right] \right] !P$$

ARGUMENTS:

adr_1

Address of the first line whose line number and contents are to be typed.

Default: Current line.

adr_2

Address of the last line whose line number and contents are to be typed.

Default: Address specified for adr_1.

NOTE

If both adr_1 and adr_2 are omitted, there is a printout of the line number and contents of the current line.

In the following examples, the contents of the current buffer are:

(1) AAA
(2) BBB (current line)
(3) CCC
(4) DDD

Example 1:

```
1,$!P
```

This Print With Line Number directive causes a printout of the line number and contents of each line in the current buffer.

Printout:

```
1 AAA
2 BBB
3 CCC
4 DDD
```

Current line: 4

Example 2:

```
!P
```

This Print With Line Number directive causes a printout of the line number and contents of only the current line.

Printout:

```
2 BBB
```

Current line: 2

UPPERCASE (!U)

Convert all occurrences of a specified expression within specified addresses from lowercase to uppercase.

After the Uppercase directive is executed, the current line is the last line read.

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] !U/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched.

Default: Current line.

adr_2

Address of the last line to be searched.

Default: adr_1.

regexp

String of characters for which the Line Editor searches. Only lowercase letters (a through z) are converted; others are not changed.

Example:

```
!U/adr/
```

This Uppercase directive searches the current line and changes each occurrence of adr to ADR. The current line is:

```
adr first
```

After the Uppercase directive is executed, the line contains:

```
ADR first
```

COMMENT

COMMENT (")

Annotate Line Editor command files. The text after the Comment directive appears as program output but is ignored by the Line Editor.

FORMAT:

```
"comment
```

AUXILIARY BUFFER DIRECTIVES AND ESCAPE SEQUENCES

In the previous pages of this section, it was assumed that there is only a single buffer, the current buffer. The current buffer must be used, but one or more additional buffers, called auxiliary buffers, also can be used. There are 64 auxiliary buffers available for use.

The most common use of auxiliary buffers is for moving or copying text from one part of a file to another.

To make an auxiliary buffer available and to put lines into it, specify the Move, Move-Append, Copy, or Copy-Append directives, which are described in the following paragraphs.

Lines cannot be written directly from an auxiliary buffer to a file; the auxiliary buffer must be designated in the Change Buffer directive as the current buffer or the lines must be read back to the current buffer via the escape sequence !B, which is described under "Change Origin of Text During Input Mode", later in this section. Lines can be written from the current buffer to a file via the Write directive (see "Write Directive" earlier in this section).

You can determine the status of each buffer currently in use by specifying the Buffer Status directive.

Auxiliary buffer directives have the following functions:

- Cause Line Editor to accept a line from terminal (Accept Single Line From a Terminal directive)
- Determine status of each buffer in use (Buffer Status directive)
- Make specified auxiliary buffer the current buffer (Change Buffer directive)
- Cause Line Editor to accept subsequent text from a specified auxiliary buffer
  - During edit mode (Change Origin of Text During Edit Mode directive)
  - During input mode (Change Origin of Text During Input Mode directive)
- Copy line(s) in current buffer to specified auxiliary buffer; lines in current buffer are not deleted
  - Delete existing lines in auxiliary buffer (Copy directive)
  - Do not delete lines in auxiliary buffer (Copy-Append directive)

- Destroy a buffer (i.e., release its file space) (Destroy directive)

- Move line(s) from current buffer to specified auxiliary buffer; lines in current buffer are deleted

  - Lines overlay existing lines, if any, in auxiliary buffer (Move directive)

  - Lines appended to existing lines, if any, in auxiliary buffer (Move-Append directive).

ACCEPT SINGLE LINE FROM A TERMINAL (!R)

Permit a single line of directives or text to be entered through a terminal. !R normally is used when Line Editor directives are being executed from a buffer. When the Line Editor encounters !R, the entire escape sequence is removed from the input stream and replaced with the line read from the user-in file.

FORMAT:

```
!R
```

Example:

```
T/ENTER YOUR NAME/
A!R!F
```

These directives are in the buffer that is being executed.

There will be the following message on the terminal:

```
ENTER YOUR NAME
```

You will respond with your name, i.e., Jane Jones.

Following the current line in the current buffer will be:

```
Jane Jones
```

BUFFER STATUS

BUFFER STATUS (X)

Display the status of each buffer currently in use. The current line is not changed.

FORMAT:

```
X
```

DESCRIPTION:

The following information is displayed:

- Name of each buffer. The original current buffer is always named 0.
- Number of lines in each buffer.
- Indicator as to which buffer is the current buffer. The name of the current buffer is preceded by ->.
- Pathname specified in the last read or write if a buffer has been read into and/or written from.

If the contents of the current buffer have been modified (i.e., in the message, MOD appears before the buffer's name), all of the following conditions must exist:

- Lines from an existing file have been read into the current buffer via a Read directive or the contents of the current buffer have been written to a file.
- The contents of the buffer were modified via one or more Line Editor directives.

Each message has the following format:

```
number of lines  ->[MOD]  (buffer-name)  [pathname]
[number of lines   [MOD]  (buffer-name)  [pathname]]
        .           .          .            .
        .           .          .            .
        .           .          .            .
```

Example:

This example illustrates usage of the buffer status directive. The file USE, which is in the working directory, comprises the following lines:

```
(1) AAA (current line)
(2) BBB
(3) CCC
(4) DDD
```

```
R USE
```

Read the contents of USE into the current buffer, which is named 0.

```
1,$S*BBB*XXX*
```

Search the first line through the last line in the current buffer and change each occurrence of BBB to XXX. After this directive is executed, the current buffer will contain:

```
(1) AAA
(2) XXX
(3) CCC
(4) DDD
```

```
3,4M2
```

Move lines 3 and 4 of the current buffer into auxiliary buffer 2. After this directive is executed, the current buffer will contain:

```
(1) AAA
(2) XXX
```

Auxiliary buffer 2 will contain:

```
(1) CCC
(2) DDD
```

```
X
```

Request the status of each buffer currently in use. The following message will be issued:

```
2  ->MOD  (0)  USE
2         (2)
```

CHANGE BUFFER (Bx)

Change a specified auxiliary buffer to the current buffer. The previously designated current buffer becomes an auxiliary buffer.

After this directive is executed, lines can be written from the new current buffer to a file.

FORMAT:

Bx

ARGUMENT:

x

Buffer name. The name must be 1 to 6 ASCII characters. If the name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional. The original current buffer name is 0. This name can never be altered. An auxiliary buffer name, once specified, cannot be altered during the current Line Editor session.

Example:

B3

This directive designates auxiliary buffer 3 as the current buffer. If desired, lines can now be written from this buffer to a file.

CHANGE ORIGIN OF TEXT DURING EDIT MODE (!B)

Cause the Line Editor to read subsequent directives from a specified auxiliary buffer. !B can be specified within an expression, pathname, text to be typed (i.e., in the Type directive), or as a directive. When the Line Editor encounters this sequence in an expression, pathname, or text, the entire escape sequence is removed from the input stream and replaced with the literal contents of the first line of the specified buffer; if !B is a directive, the input stream is replaced with the entire literal contents of the specified buffer. If another !B escape sequence is encountered while accepting input from buffer x, the newly encountered escape sequence will also be replaced by the contents of its named buffer.

The buffer to which the input stream is redirected may contain Line Editor requests, literal text, or both. If the Line Editor is executing a request obtained from an auxiliary buffer and an error occurs, the usual error comment is suppressed and the remaining contents of that buffer are skipped. Control returns to the statement immediately following the !B escape sequence that called the auxiliary buffer. For example, if one thinks of the escape sequence !B(X) as a subroutine call statement, the failure to match a regular expression specified by some request in buffer x may be thought of as a return statement. Once the last commands in the auxiliary buffer have been processed, control returns to the statement immediately following the !B escape sequence that called the auxiliary buffer.

The buffer name may be in the format (ARGn), where n is a number from 1 to 9 that refers to the nth argument that followed the -ARG argument of the ED command. The escape sequence is replaced with the first (or only) line of the buffer (ARGn) created during initialization of the Line Editor.

FORMAT:

```
!Bx
```

ARGUMENT:

x

Name of the buffer that contains subsequent Line Editor text. The buffer name must be 1 through 6 ASCII characters. If the buffer name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example 1: !B as a directive

```
!B(TEST)
```

In this example, the contents of the current buffer and the auxiliary buffer named TEST are:

Current buffer:

```
(1) A
(2) B
(3) A
(4) D
(5) E
```

Auxiliary buffer:

```
1,$S/A/X/
```

This Substitute directive requests that in the current buffer all occurrences of A be replaced with X. After the Substitute directive is executed, the current buffer will contain:

```
(1) X
(2) B
(3) X
(4) D
(5) E
```

The auxiliary buffer named TEST will contain:

```
1,$S/A/X/
```

Example 2: !B Within an Expression

```
2S/AAA/!B2/
```

This Substitute directive requests that in the second line of the current buffer, each occurrence of AAA should be replaced with the first line of auxiliary buffer 2.

The contents of the current buffer and auxiliary buffer 2 are:

Current buffer:

```
(1) AAABBB
(2) CCCAAA
(3) XXXYYY
```

Auxiliary buffer 2:

```
DDD
EEE
```

After the Substitute directive is executed, the current buffer contains:

```
(1) AAABBB
(2) CCCDDD
(3) XXXYYY
```

Example 3: !B Within Text to be Typed

```
T/!B2/
```

This Type directive (which is described later in this section) requests that the first line of auxiliary buffer B2 be displayed on the user-out file.

Example 4: Buffer Name (ARGn)

The ED command includes the argument -ARG ABC "MY NAME" XYZ

```
S/DEF/!B(ARG3)/
```

This Substitute directive searches the current line and replaces each occurrence of DEF with XYZ (i.e., the third argument following -ARG in the ED command).

CHANGE ORIGIN OF TEXT DURING INPUT MODE

CHANGE ORIGIN OF TEXT DURING INPUT MODE (!B)

Cause the Line Editor to accept subsequent text from a specified auxiliary buffer. The escape sequence !B can appear within text of an Input directive.

When the Line Editor encounters !B, the entire escape sequence is removed from the input stream and replaced with the literal contents of the specified buffer. If another !B escape sequence is encountered after accepting text from the specified buffer, the newly encountered escape sequence will also be replaced with the contents of the named buffer.

FORMAT:

```
{text}!Bx [{text}!B] ...
```

ARGUMENT:

x

Name of the buffer that contains subsequent Line Editor text. The buffer name must be 1 through 6 ASCII characters. If the buffer name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example:

```
/D/I
!B(TEST)!F
```

In this example, the contents of the current buffer and the auxiliary buffer named TEST are:

Auxiliary buffer:

```
(1) X
(2) Y
(3) Z
```

Current buffer:

```
(1) A
(2) B
(3) C
(4) D
(5) E
```

This Insert directive inserts the contents of the auxiliary buffer named TEST into the current buffer before the line that contains D.

After the Insert directive is executed, the current buffer will contain:

```
(1) A
(2) B
(3) C
(4) X
(5) Y
(6) Z
(7) D
(8) E
```

The auxiliary buffer named TEST will contain:

```
(1) X
(2) Y
(3) Z
```

COPY (K)

Write into a specified auxiliary buffer a single line or consecutive lines contained in the current buffer. The lines in the current buffer are <u>not</u> deleted; i.e., the lines are in both the current and the auxiliary buffers. Any lines previously in the auxiliary buffer are destroyed during execution of the Copy directive.

After the Copy directive is executed, the current line in the current buffer is the last line moved to the auxiliary buffer. There is no current line in the auxiliary buffer until that auxiliary buffer is changed to the current buffer via a change buffer directive.

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] Kx$$

ARGUMENTS:

adr_1

Address of the first line to be written into the specified auxiliary buffer.

Default: Current line.

adr_2

Address of the last line to be written into the specified auxiliary buffer.

Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is written into the specified auxiliary buffer.

x

Name of the auxiliary buffer into which the specified line(s) will be written. The name must be 1 through 16 ASCII characters. If the name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example:

```
1,3K(52)
```

This Copy directive copies into auxiliary buffer 52 lines 1 through 3 in the current buffer. The contents of the current buffer are:

```
(1) FIRST (current line)
(2) SECOND
(3) THIRD
(4) FOURTH
```

After the Copy directive is executed, the contents of the current buffer are unchanged, but the current line is line number 4. Auxiliary buffer 52 will contain:

```
(1) FIRST
(2) SECOND
(3) THIRD
```

There will be no current line in the auxiliary buffer.

COPY-APPEND (1K)

Write a line or lines from the current buffer to an auxiliary buffer without destroying the contents of the auxiliary buffer. The lines copied from the current buffer are appended to the contents of the auxiliary buffer. The lines written are also retained in the current buffer.

After the Copy-Append directive is executed, the current line in the current buffer is the the last line written to the auxiliary buffer or the last line in the buffer. There is no current line in the auxiliary buffer.

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] !Kx$$

ARGUMENTS:

adr_1

Address of the first line to be written to the specified auxiliary buffer.

Default: Current line.

adr_2

Address of the last line to be written to the specified auxiliary buffer.

Default: adr_1.

NOTE

If both addresses are omitted, only the current line is written to the auxiliary buffer.

x

Name of the auxiliary buffer into which the specified line(s) will be written. The name must be from 1 to 16 ASCII characters. If the name is more than one character, it must be enclosed within parentheses; otherwise, parentheses are optional.

Example:

```
1,3!K(ABUF)
```

This directive appends lines 1 through 3 of the current buffer to the contents of auxiliary buffer ABUF. Thus, if the current buffer and ABUF contain the following lines prior to execution:

Current	ABUF
(1) AAA (current line)	(1) MMM
(2) BBB	(2) NNN
(3) CCC	
(4) DDD	

They will contain the following after execution:

Current	ABUF
(1) AAA	(1) MMM
(2) BBB	(2) NNN
(3) CCC	(3) AAA
(4) DDD (current line)	(4) BBB
	(5) CCC

DESTROY (^B)

Release a specified auxiliary buffer's file space. Any buffer other than buffer 0 and the current buffer may be removed; if the current buffer name is specified, the directive is ignored and an error message is issued.

FORMAT:

^Bx

ARGUMENT:

x

Name of the auxiliary buffer to be destroyed. The name must be from 1 to 6 ASCII characters. If the name comprises more than one character, it must be enclosed within parentheses; otherwise, parentheses are optional.

Example:

^B(AX)

This Destroy directive removes buffer AX.

MOVE (M)

Move a single line or consecutive lines from the current buffer to a specified auxiliary buffer; the lines no longer exist in the current buffer. Any lines already in the auxiliary buffer are destroyed by the Move operation.

After the Move directive is executed, the current line in the current buffer is the line after the last line moved to the auxiliary buffer or the last line in the buffer. There is no current line in the auxiliary buffer.

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] Mx$$

ARGUMENTS:

adr_1

Address of the first line to be moved from current buffer to auxiliary buffer.

Default: Current line.

adr_2

Address of the last line to be moved from current buffer to auxiliary buffer.

Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is moved from the current buffer to the auxiliary buffer.

x

Name of the auxiliary buffer to which the specified line(s) will be moved. The name must be 1 through 6 ASCII characters. If the name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example:

```
1,3M5
```

This Move directive moves lines 1 through 3 from the current buffer to the auxiliary buffer named 5. In this example, the contents of the current buffer are:

```
(1) FIRST (current line)
(2) SECOND
(3) THIRD
(4) FOURTH
```

After the Move directive is executed, the current buffer will contain:

```
(1) FOURTH (current line)
```

Auxiliary buffer 5 will contain:

```
(1) FIRST
(2) SECOND
(3) THIRD
```

MOVE-APPEND (!M)

Move one or more lines of text from the current buffer to the specified auxiliary buffer. The lines are appended to the existing contents of the auxiliary buffer; the existing contents of the auxiliary buffer are not destroyed. If the auxiliary buffer contains no text, the lines are placed in the auxiliary buffer starting at line 1. The lines moved are deleted from the current buffer.

FORMAT:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] !Mx$$

ARGUMENTS:

adr_1

Address of the first line to be moved from the current buffer to the auxiliary buffer.

Default: Current line.

adr_2

Address of the last line to be moved from the current buffer to the auxiliary buffer.

Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is moved from the current buffer to the auxiliary buffer.

x

Name of the auxiliary buffer to which the specified line(s) will be moved. The name must be 1 through 6 ASCII characters. A name of more than one character must be enclosed in parentheses; otherwise, parentheses are optional.

Example:

```
1,3!M(SOOZ)
```

This directive appends lines 1 through 3 to the contents of auxiliary buffer SOOZ. The contents of the buffers are as follows prior to the move:

Current	SOOZ
(1) FIRST (current line)	(1) AAAAA
(2) SECOND	(2) BBBBB
(3) THIRD	
(4) FOURTH	

The buffers will contain the following after the move:

Current	SOOZ
(1) FOURTH (current line)	(1) AAAAA
	(2) BBBBB
	(3) FIRST
	(4) SECOND
	(5) THIRD

LINE EDITOR DEBUGGING DIRECTIVES

The functions of Line Editor debugging directives are:

- Print contents of specified line(s) on the terminal (Hexadecimal Dump directive)
- Display, on the user-out file, the last specified regular expression (ZREGEXP directive)
- Display each directive line before it is executed (ZTRACE directive).

HEXADECIMAL DUMP (ZDUMP)

Print the contents of specified line(s) on the terminal in both hexadecimal and ASCII formats. The output format consists of the line number, the length (number of characters) expressed in hexadecimal, eight words in hexadecimal format, and eight words in ASCII format.

The display of each buffer line is separated from following displays by a blank line. If a buffer line is too long to be displayed on a single line, it is continued on the next line, with no blank line separation.

After this directive is executed, the current line is the last (or only) line printed.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2 \right] \right] \text{ZDUMP}$$

ARGUMENTS:

adr_1

Address of the first buffer line to be dumped.

Default: Current line.

adr_2

Address of the last buffer line to be dumped.

Default: adr_1.

NOTE

If both addresses are omitted, only the current line will be dumped.

Example:

The contents of lines 1 and 2 of the current buffer are:

```
(1) START EDIT
(2) VDEF ZFVER,X'3031'

1,2ZDUMP
```

This Hexadecimal Dump directive produces the following output at the terminal:

```
0001 000A 5354 4152 5420 4544 4954                    START EDIT

0002 0012 5644 4546 205A 4656 4552 2C58 2733 3033 VDEF ZFVER,X'303
          3127                                     1'
```

Thus, 0001 indicates line number 1; 000A indicates a length of 10 characters (A_{16}); followed by the hexadecimal equivalent of START EDIT. A blank line is followed by the dump of line 2, with a length of 18 characters (12_{16}). Because nine words are required to fully dump the line, the output continues on the next line of the terminal, with no blank line intervening.

ZREGEXP

Display the last specified expression on the user-out file. The current line is not changed.

FORMAT:

```
ZREGEXP
```

Example:

```
S/ABC/DEF/
ZREGEXP
```

This ZREGEXP directive displays the last specified expression, i.e., /ABC/.

ZTRACE

Display each directive line on the user-out file before it is executed.

FORMAT:

ZTRACE { ON / OFF }

ARGUMENTS:

ON Each directive line is displayed before it is executed.

OFF Subsequent lines are not displayed before they are executed.

Example:

This example illustrates a program that includes an ED command to load the Line Editor and a ZTRACE ON directive. Following is a printout of the Line Editor output.

Program including ED command and ZTRACE ON directive:

```
 1  RL DIRECTORY
 2  FO DIRECTORY
 3  WS &1 "LS -BF"
 4  FO
 5  &A
 6  ED
 7  ZTRACE ON
 8  B1
 9  I
10  R DIRECTORY
11  GD/^ &/
12  GD/^.   ENTRY NAME  TYPE$/
13  GD/ D$/
14  1,$S/^.   //
15  1,$S/^DIRECTORY:  . //
16  $N
17  :C ?/^^/;M(2)
18  :D ^*/^^/S/^.*$/&   !C!B2>&/?+1;>D
19  ?+1,-1N>C
20  */^^/D!F
21  B0
22  !B1
23  W DIRECTORY
24  Q
```

Line Editor output:

```
 EDIT-0200-09/11/0948
**EDIT**  B1
**EDIT**  I
**INPUT** R DIRECTORY
**INPUT** GD/^ $/
**INPUT** GD/^.    ENTRY NAME  TYPE$/
**INPUT** GD/ D$/
**INPUT** 1,$S/^.    //
**INPUT** 1,$S/^DIRECTORY:   . //
**INPUT** $N
**INPUT** :C ?/^^/;M(2)
**INPUT** :D ^*/^^/S/^.*$/&!C!B2>&/?+1;>D
**INPUT** ?+1,-1N>C
**INPUT** */^^/D!F
**EDIT**  B0
**EDIT**  !B1
**EDIT**  R DIRECTORY
**EDIT**  GD/^ $/
**EDIT**  GD/^.    ENTRY NAME  TYPE$/
**EDIT**  GD/ D$/
**EDIT**  1,$S/^.    //
**EDIT**  1,$S/^DIRECTORY:   . //
**EDIT**  $N
**EDIT**  :C ?/^^/;M(2)
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1:>D
**EDIT**  ?+1,-1N>C
**EDIT**  :C ?/^^/;M(2)
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  ?+1,-1N>C
**EDIT**  :C ?/^^/;M(2)
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  ?+1,-1N>C
**EDIT**  */^^/D
**EDIT**  W DIRECTORY
**EDIT**  Q
```

LINE EDITOR PROGRAMMING DIRECTIVES

Line Editor programming directives cause conditional execution of subsequent directives, change the location of subsequent Line Editor input, and display a line of text on the user-out file. Programming directives can be in the directive input file (specified in the -IN path argument of the ED command) or an auxiliary buffer, or they can be entered through a terminal.

Each conditional directive includes one or more other Line Editor directives. The directives must be on a single line. If the specified condition exists, the subsequent embedded directive(s) are executed. The following conditions can be tested:

- Does specified line exist (Address Prefix directive)
- Does current buffer contain data (If Empty and If Data directives)
- Is current line a specified line (If Line and If Not Line directives)
- Is current line within specified lines (If Range and If Not Range directives)
- Is specified expression within specified lines (Search and Search Not directives).

Programming directives also have the following capabilities:

- Change location from which Line Editor accepts subsequent directives (Go To directive)
- Define location that can be the endpoint of a Go To directive (Label directive)
- Display a line of text on the user-out file (Type directive).

NOTE

If a directive format comprises multiple directives, the directives may be separated by spaces for readability.

ADDRESS PREFIX (?)

Execute the directives contained in the Address Prefix Line if the specified line exists in the current buffer; otherwise, do not execute them.

FORMAT:

?adr{;,}directive [directive] ...

ARGUMENTS:

adr

Address of the line for which the Line Editor will search.

NOTE

If adr is immediately followed by a semicolon, adr becomes the current line. If adr is immediately followed by a comma, the current line is not changed.

directive

Any Line Editor directive(s); they are executed only if the specified line is found.

Example 1:

?8;P

This Address Prefix directive specifies that if there is a line 8 in the current buffer, print the contents of that line; that line will become the current line.

Example 2:

In this example, the contents of the current buffer are:

(1) DEFGHI
(2) ABCXYZ
(3) ABCGGG (current line)

?/ABC/;S/ABC/DEF/

This Address Prefix directive designates that if there is a line that contains ABC, make that line the current line, and in that line replace each occurrence of ABC with DEF.

After this directive is executed, the current buffer will contain:

(1) DEFGHI
(2) DEFXYZ (current line)
(3) ABCGGG

GO TO

GO TO (>)

Change the location from which the Line Editor accepts subsequent directives.

If the Go To directive is encountered in the buffer that is currently being executed, the Line Editor accepts subsequent directives from a specified location in that buffer. The location must have been previously defined in that buffer by a Label directive.

If the Go To directive is entered interactively, only directives in the current directive line are used.

FORMAT:

```
>label
```

ARGUMENT:

label

Location to which control is transferred; the Line Editor accepts subsequent directives from this location.

If the label comprises multiple characters, they must be enclosed within parentheses; otherwise, the parentheses are optional.

Example 1:

In this example, the contents of the current buffer are:

```
(1) EAST ROCKAWAY, NY
(2) LONG BEACH, NY
(3) BRIGHTON, MASS
(4) ANDOVER, MASS
(5) HEWLETT, NY
```

Buffer 2 contains the following directives:

```
:(REPEAT)1,$P
```

Assign label REPEAT to Print directive line.

```
1,$S/MASS$/MASSACHUSETTS/P
```

Substitute each occurrence of MASS at the end of a line with MASSACHUSETTS and print the contents of the last line in the buffer (i.e., line number 5).

NOTE

When the Line Editor searches the buffer the second time and does not find MASS at the end of a line, control returns to the previous buffer or to the terminal.

1,$S/NY/NEW YORK/>(REPEAT)

Substitute each occurrence of NY with NEW YORK and print the contents of all lines (i.e., lines 1 through 5).

Example 2:

:A?/ABC/;S/ABC/DEF/P>A

If this directive is entered interactively, the following actions take place. The information to the right of each action indicates how the action is requested in the directive line.

Assign label A to directive line.	:A
If ABC exists, take the subsequent actions.	?/ABC/
Change the current line to the location of ABC.	; preceding the substitute directive
Replace each occurrence of ABC with DEF.	S/ABC/DEF/
Print the current line.	P
Go to line A (i.e., reexecute the same directive line)	>A

After all lines containing ABC have been acted upon (i.e., each occurrence of ABC has been replaced with DEF and the resulting lines printed), control returns to the next directive entered interactively.

IF DATA

IF DATA (#)

Execute the directives contained on the If Data directive line if the current buffer contains data; otherwise, do not execute them.

FORMAT:

```
#directive [directive] ...
```

ARGUMENT:

directive

Any Line Editor directive(s); they are executed only if the current buffer contains data.

IF EMPTY (^#)

Execute the directives contained in the If Empty directive line if the current buffer is empty; otherwise, do not execute them.

FORMAT:

^#directive [directive] ...

ARGUMENT:

directive

Any Line Editor directive(s); they are executed only if the current buffer does not contain data.

IF LINE

IF LINE (adr#)

Execute the directives contained on the If Line Directive line if the current line is the specified line; otherwise, do not execute them.

FORMAT:

adr#directive [directive] ...

ARGUMENTS:

adr

Address of the line being checked to see if it is the current line.

directive

Any Line Editor directive(s); they are executed only if the specified line is the current line.

IF NOT LINE (adr^#)

Execute the directives on the If Not Line directive line if the current line is not the specified line; otherwise, do not execute them.

FORMAT:

```
adr^#directive [directive] ...
```

ARGUMENTS:

adr

Address of the line being checked to see if it is the current line.

directive

Any Line Editor directive(s); they are executed only if the specified line is not the current line.

IF RANGE

IF RANGE (adrs#)

Execute the directives on the If Range directive line if the current line is within specified lines; otherwise, do not execute them.

FORMAT:

$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2$ #directive [directive] ...

ARGUMENTS:

adr_1 :

Address of the first line to be searched.

adr_2 .

Address of the last line to be searched.

directive

Any Line Editor directive(s); they are executed only if the current line is within addresses adr_1 through adr_2. The current line is unchanged.

IF NOT RANGE (adrs^#)

Execute the directives on the If Not Range directive line if the current line is not within specified lines; otherwise, do not execute them.

FORMAT:

adr_1 { ; , } adr_2^#directive [directive] ...

ARGUMENTS:

adr_1

Address of the first line to be searched.

adr_2

Address of the last line to be searched.

directive

Any Line Editor directive(s); they are executed only if the current line is not within addresses adr_1 through adr_2. The current line is unchanged.

Example:

```
1,10^#S/yes/no/
```

This If Not Range directive specifies that if the current line is not within lines 1 through 10, in the current line substitute each occurrence of "yes" with "no".

SEARCH (*)

Execute the directives on the Search directive line if a specified expression is within specified lines; otherwise, do not execute them.

FORMAT:

$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2$*/regexp/directive [directive] ...

ARGUMENTS:

adr_1

Address of the first line to be searched for the regular expression.

Default: Current line.

adr_2

Address of the last line to be searched for the regular expression.

Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is searched.

regexp

String of characters for which the Line Editor is searching.

directive

Any Line Editor directive(s); they are executed only if the specified expression is within the specified addresses.

SEARCH NOT (^*)

Execute the directives on the Search Not directive line if a specified expression is not within specified lines; otherwise, do not execute them. The current line is unchanged.

FORMAT:

$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2$ ^*/regexp/directive [directive] ...

ARGUMENTS:

adr_1

Address of the first line to be searched for the regular expression.

Default: Current line.

adr_2

Address of the last line to be searched for the regular expression.

Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, the directives are executed only if the regular expression is not in the current line.

regexp

String of characters for which the Line Editor is searching.

directive

Any Line Editor directive(s); they are executed only if the specified expression is not within the specified addresses. The current line is unchanged.

LABEL

LABEL (:)

Define a location to which the Line Editor can be directed (via a Go To directive) for subsequent directives. If a Go To directive is entered interactively, only the current directive line is searched for the label. The Label directive must be specified at the beginning of a line.

FORMAT:

```
:labeldirective [directive] ...
```

ARGUMENTS:

label

Location that can be the argument value of a Go To statement (i.e., a location to which control can be transferred). If multiple characters constitute the label, they must be enclosed within parentheses; otherwise, parentheses are optional.

directive

Any Line Editor directive(s); they are executed when control passes to the specified label.

TYPE (T)

Display a line of text on the user-out file. If the optional exclamation point (!) is specified in the directive line, the next input or output will appear immediately after the printout, on the same line; otherwise, the next printouts are on subsequent lines.

FORMAT:

[!]T/text/

ARGUMENTS:

/

(Delimiter) Can be any nonblank character, but the same character must be used in each place where a delimiter is required.

text

Text to be displayed. Default: One blank line.

Example 1:

T/IDENTIFICATION NUMBER/

This Type directive prints IDENTIFICATION NUMBER. Since the optional exclamation point was not specified, subsequent input or output will appear on subsequent lines.

Example 2:

!T/IDENTIFICATION NUMBER !B2/

This Type directive prints IDENTIFICATION NUMBER and the contents of auxiliary buffer B2. If B2 contains FOR THIS YEAR, the printout will be: IDENTIFICATION NUMBER FOR THIS YEAR. Since the directive name T was immediately preceded by an exclamation point, the next input or output will appear immediately after the printout, on the same line.

PROGRAMMING CONSIDERATIONS

1. Tabbing causes embedded tab characters to be replaced with the appropriate number of spaces so that printed output on a printer or terminal has "tab stops" at character position 11 and at every subsequent 10 character positions. Tab characters can be entered into Assembly language source lines by pressing CTRL I on the terminal device while entering insert and/or substitute directive(s). CTRL I is a nonprinting tab character that has a hexadecimal value of 09. Tabbing is not apparent until a printout occurs.

2. The Line Editor uses a minimum of two temporary work files in the working directory. These files are created by the Line Editor when the Line Editor is invoked; they exist only during the current execution of the Line Editor. A minimum of 16 diskette or 8 cartridge sectors must be available in the working directory for temporary work files. Additional temporary files are created for each auxiliary buffer used; the number of temporary files is limited by the space available in the working directory.

3. If you specify a buffer name comprising more than a single character and omit the parentheses, only the first character is considered the buffer name; subsequent characters are treated as directives.

4. If a file manager error (190223, lack of space) or a physical I/O error (190107) is encountered, use the Quit directive to exit from the Line Editor, and restart after the problem has been corrected. Attempting to recover by other means (such as the escape sequences) may cause unspecified results. If an error occurs while processing a work file (this situation is indicated by an error message that is not followed by a file name), the Line Editor may terminate processing and a fatal error message is issued.

5. An error occurs if the maximum number of lines that the Line Editor will accept in a program has been reached. Control is returned to command level.

Section 16
LINKER

OVERVIEW

The Linker combines object units created by the language processors (compilers and the Assembler) into a bound unit that you can then execute. During a single execution of the Linker, a single bound unit is created. A bound unit contains a root or a root with one or more overlays. The root and overlays cannot exceed the physical memory available in your system's configuration.

LINKER FUNCTIONS

The Linker functions are:

- CREATE A BOUND UNIT -- A bound unit is the output file that results from Linker execution. The bound unit is an executable program.
- BUILD A SYMBOL TABLE -- During the linking process, the Linker builds an internal symbol table used for resolving external references. You can define a symbol within an object unit or by using Linker directives defined later in this section.

- PRODUCE A LISTING -- The linker listing has two parts, a dynamic part and a static part.

  - The dynamic part is generated continuously and contains information about each object unit linked, the directives used, and a summary.

  - The static part is produced in response to the MAP or MAPU directive and is a picture of the state of the link when the MAP(U) directive is processed. It lists the external definitions currently in the symbol table and the undefined external references, if any exist.

  During the link process, summary information about the bound unit is automatically output to a list file. The format of this information is:

```
**********
ROOT  TESTP2
* HIGHEST OVERLAY NUMBER:  2
 LAF
**********
* * CMMN  DATA              BASE:  000000  START:  000000  .F..  HIGH:  000011
  ROOT  TESTP2             BASE:  000000  START:  000000  ..U.  HIGH:  00003F
  † OVLY  OVLNO      # 0001 BASE:  00003F  START:  00003F  ....  HIGH:  000060

KEY:  S=SHAREABLE; F=FLOATING; I=CONTAINS AN IMA; U=CONTAINED AN UNDEFINED
      REFERENCE; ->=IN-LINE DIRECTIVE; [...]=EMBEDDED DIRECTIVE

**********
SIZE OF ROOT AND FIXED OVERLAYS:  000060
LAST BU RECORD NUMBER:  4
**********
LINK DONE
**********
```

- RESOLVE EXTERNAL REFERENCES -- The Linker resolves addresses or values of external symbol references in object units being linked. To do this, the Linker uses external definitions found in the object units or declared by the LDEF or VDEF directives. (LDEF and VDEF are described fully later.) When a bound unit is linked, the unresolved external references are listed at the end of the link map. If unresolved external references exist at the end of the list, an error message is displayed on the error-out file, usually the terminal.

---

* Each control interval (logical record on the bound unit file) has a size of 256 bytes (128 words).

**This line only appears if common has been gathered into one contiguous area. The -R ECL parameter was specified.

†This line repeated for each overlay.

LINKER DIRECTIVE CATEGORIES

The Linker directive set may be grouped into nine functional categories described in the following paragraphs.

Specifying Object Unit(s) to be Linked

LINK, LINKN, LINKnn, and LINKO designate that one or more specified object units are to be linked. Object units specified in LINK directives are not linked immediately; their names are put into a link request list. Once a directive has been entered which requires that all preceding link requests be honored, linking begins. Specified object units in the primary input directory are linked before specified object units in the secondary input directory; within each directory, the object units are linked in the order in which they were requested.

LINKN causes the Linker to link object units already named in the link request list, and then to link object units specified in the LINKN directive in the order in which they were requested.

LINKO performs in the same manner as LINKN, except that all embedded directives in the named object unit(s) are ignored by the Linker. LINKnn is a special form of LINKN used to perform selective linking.

Specifying Location(s) of Object Unit(s) to be Linked

Object units to be linked must be in at least one directory. The Linker searches the primary directory first, then searches other directories if they have been specified by directives described below. When the Linker is loaded into memory, the primary directory is the working directory. The directives used to specify location(s) of object unit(s) to be linked are listed below.

IN is used to designate a directory other than the working directory as the primary directory.

LIB is used to designate a directory as the second directory to be searched.

LIB2 is used to designate the third directory to be searched.

LIB3 is used to designate the fourth directory to be searched.

LIB4 is used to designate the fifth directory to be searched.

LSR is used to request a list of the directories in the order in which they are to be searched.

Creating a Root and Optional Overlay(s)

START is used to specify the relative address at which the root or overlay will begin executing when it is loaded into memory by the Loader.

BASE is used to define relative addresses (within the bound unit) for subsequent object units to be linked. Note that when the lowest address of a root or overlay has been established (i.e., an object unit has been linked), it is invalid to define a lower BASE address within the root or overlay.

OVLY is used to name the nonfloatable overlay that follows, and designates the end of the preceding root or overlay.

FLOVLY is used to name the floatable overlay that follows, and designates the end of the preceding root or overlay.

CC permits a COBOL program that used CALL and CANCEL statements to call overlays by their names.

IST is used to identify the beginning of initialization code in the root.

SHARE is used to designate that the bound unit is sharable within the task group.

QUIT is used to designate that the last Linker directive has been entered. Execution of the Linker terminates after the bound unit has been created.

FLOATB6 is used to suppress certain error checking on local common references when the -R Linker argument has not been specified. Local common references are relocated as if B6 pointed to the base of the containing overlay.

STACK is used to specify the size of the stack area.

GSHARE is used to specify that the bound unit is globally sharable.

SEG is used to specify that the subsequent object unit is to be linked into one or two physical segments in memory.

SYS is used to designate that the bound unit can be loaded into the system area as part of the system.

LINK, LINKN, and LINKO are used to specify those object units to be linked. The order in which specified object units are linked, and when they are linked, is determined by the link directive used.

LOADING THE LINKER

The command LINKER is used to load the Linker.

After the Linker is loaded, a message is sent to the error-out file indicating the version. The message format is:

LINKER-nnnn-mm/dd/hhmm

where nnnn is a release identification, mm/dd is the month and day the Linker component was linked, and hhmm the time (hour, minutes) at which that link took place.

FORMAT:

LINKER bound-unit-path [ctl_arg]

ARGUMENTS:

bound-unit-path

Pathname of the bound unit file. The pathname can be simple, relative, or absolute and must be preceded by a space. If the specified file already exists, the existing information in the file is deleted and replaced with the new bound unit. The bound unit pathname must be specified. It may be up to 57 characters in length. The format of the bound unit file is relative.

ctl_arg

Control arguments; none or any number of the following control arguments can be entered, in any order:

{-IN, -I} path

Pathname of the device disk, card reader, operator's terminal, or another terminal that will read Linker directives.

Default: Device specified in the in_path argument of the Enter Group Request command.

When this argument is specified, the prompt character will not appear.

PURGE is used to remove from the symbol table unprotected symbols that define a specified address or an address within a specified range, and/or object unit names equated to a specified address or an address within a specified range.

VPURGE is used to remove a specified value definition from the symbol table.

Reloading After System Failure

RR indicates that a sharable bound unit can be reloaded after a system failure into locations other than those it occupied at checkpoint.

Controlling the Directive File

The user specifies by the -IN argument of the LINKER command the user-in file, from which the Linker reads directives.

An INCLUDE directive causes the Linker to accept directives from a file specified with the directive rather than from user-in.

When the Linker encounters a RETURN directive in the file specified with INCLUDE, the Linker returns to user-in.

Terminating the Linker

QUIT is used to terminate the Linker. If a bound unit is being created, execution of the Linker terminates after the bound unit has been created. If no bound unit is being created, QUIT terminates execution of the Linker.

Subsections that follow include full information on:

Loading the Linker -- Describes the Linker command used to call the Linker and initiate Linker processing.

Entering Linker Directives -- Describes the format line used to enter directives.

Linker Directive Set -- Provides an alphabetic listing of the Linker directives. Detailed descriptions of each directive and examples of use are provided.

Linker Procedures -- Describes frequently used Linker procedures.

Producing Link Map(s)

LDEF is used to assign a relative location to an external symbol. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used to resolve references to the symbol during linking.

VDEF is used to assign a value to an external symbol. When a symbol is defined, its definition is put into the linker symbol table so that it can be used during linking to resolve external references.

MAP is used to create a map that lists both defined and undefined symbols.

MAPU is used to create a map that lists the undefined symbols only.

-V ECL option will automatically list symbols as they are defined.

Defining External Symbols

A symbol can be defined as a relative location or value by specifying the LDEF or VDEF directive, respectively. The symbol's definition is then put into the symbol table by the Linker.

EDEF permits definitions in the Linker symbol table to be made part of the bound unit so that they are available to the Loader at execution time.

OVERLAYTABLE is used to put a value definition containing the name of each overlay and its overlay number in the bound unit symbol table.

COMM is used to define a labeled common block.

VAL is used to specify a value definition at LINK time. This value is equivalent to the difference between two external location definitions.

Protecting or Purging Symbol(s)

CPROT and CPURGE are used to protect and remove symbols associated with labeled common blocks.

PROT and PURGE are used to protect and remove symbols and object unit names from the symbol table. PROT prevents certain symbols and/or object unit names from being removed from the symbol table. Symbols are protected if they identify a specified address or an address within a specified range; object unit names are protected if they are equated to a specified address or an address within a specified range.

-PT

If the -IN argument is not specified, -PT can be specified to produce a prompt character on the user terminal. A prompt character is issued only if -PT is specified.

{-COUT / -COUTA} list-path-name

Name of the list file. The list file can be sent to a disk, another terminal, or a printer. The list-path-name is associated with this list file. If -COUT is not specified, the list-path-name has a default value of bound-unit-name.M in the working directory. If -COUTA is specified, the listing is appended to the specified file.

Error messages are written to the error-out file and the list file. Linker error mesages are described in the System Messages manual.

{-SIZE nn / -S%}

nn designates the maximum number of 1024-word (1K) blocks of memory available for the Linker symbol table; nn must be from 1 to 64. At least 1024 words must be available.

Default: 2

-W

Save the Linker work files.

Default: Linker work files are automatically released by the Linker upon Linker termination.

-R

Create a bound unit where all data areas defined as common are separated from all other code. Required for sharable bound units containing common data areas.

{-VERBOSE / -V}

Write externally defined symbols on the list file as they are defined. Eliminates the need for the MAP directive.

-NOMAP

Suppress the list file.

{ -SYMBOL }
{ -SYM   }

Create a debugger information file. This file is used for symbolic debugging. The name of the file is buname.v. This option should only be used for FORTRANA or COBOLA programs.

Example:

```
LINKER MYPROG -IN MYDISK>CNL -COUT !LPT00 -SIZE 6
```

This LINKER command loads the Linker and specifies the following:

1. Bound unit will be a relative file named MYPROG in the working directory.

2. Linker directives will be entered through disk file MYDISK>CNL.

3. List file goes to a line printer (configured as LPT00), rather than to a variable sequential file named MYPROG.M in the working directory.

4. The symbol table will use a maximum of 6K words of memory.

NOTE

LPT00 must have been previously defined in the DEVICE configuration directive at system generation time.

ENTERING LINKER DIRECTIVES

Linker directives are entered through the directive input device. Several directives can also be embedded in Assembly language CTRL statements. They are: LINK, LINKN, LINKO, SHARE, EDEF, SYS, COMM, LSR, and VAL.

Linker directives consist of a directive name or a directive name followed by one or more arguments. Each directive name may be preceded by zero or more blank spaces. If one or more arguments are to be specified in a Linker directive, the directive name must be immediately followed by one or more spaces.

Multiple directives can be entered on a line by specifying a semicolon (;) after each directive except the last on the line.

The last directive on a line can be followed by a comment; to include a comment, specify a space and a slash (/) after the last directive and then enter the comment.

FORMAT:

directive [Δargument$_1$] [Δargument$_2$] [Δ/comment]

If the directive input device is the operator's terminal or another terminal, press RETURN at the end of each line (i.e., at the end of the comment, or at the end of the last directive if there is no comment). There is no continuation between lines; the values associated with a single directive cannot be continued on a second line.

If an error occurs when entering a directive, an error message is written to the error-out file. Linker error messages are described in the System Messages manual. Determine what caused the error, and reenter the directive correctly. If multiple directives are entered on a line and an error occurs, the error does not affect the execution of previously designated directives. The directive that caused the error and subsequent directives on that line are not executed.

LINKER DIRECTIVES SET

Linker directives are described in alphabetic order on the following pages. Examples are provided to illustrate directive usage.

BASE

Defines the relative link address within the bound unit for subsequent object units to be linked. At load time, all addresses are relative to the beginning of available memory (relative 0) in the memory pool of the task group. When a task group is created, you specify the memory pool into which its bound units are to be loaded.

Unless BASE directives specify otherwise, the root will be linked, by default, at relative 0, and subsequent object units are linked at successive relative addresses. A BASE directive can be used at any point during linking to change the relative locations of the root, overlays, or individual object units. A floatable overlay always begins at relative 0; therefore, in a floatable overlay, BASE can be specified only after the first LINK, LINKN, or LINKO directive. A BASE directive can specify a previously used or defined location, or an address relative to the beginning of the available memory.

If unprotected symbols define locations that are equal to or greater than the location designated in the BASE directive, those symbols are removed from the symbol table.

The BASE directive cannot be embedded in Assembly language control statements.

FORMAT:

```
        $
        %
        X'address'
        =object-unit-name
BASE    xdef [{±}X'offset']
        #
        *ODD
        *EVEN
        *X'offset'
```

ARGUMENTS:

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

$

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

A one- to five-character hexadecimal address enclosed in single quotation marks and preceded by X. The specified address is relative to the beginning of the root (relative 0).

=object-unit-name

Specified object unit's base address; the subsequent root, overlay, or object unit will be linked at the same relative address as the specified object unit, which must have already been linked. Furthermore, the object unit name must still exist in the symbol table (i.e., it has not been purged).

xdef[{±} X'offset']

Address of any previously defined (non-common) external symbol. If an offset is specified, it must be a hexadecimal integer with an absolute value less than 8000 (32768 decimal).

#

The current address.

*ODD

The current address, if it is odd; if it is even, base address is converted to current address+1.

*EVEN

The current address, if it is even; if it is odd, base address is converted to current address+1.

*X'offset'

The next location whose rightmost hexadecimal characters equal the offset (where the offset is a hexadecimal integer of four or fewer characters).

Default: $ with the following exceptions:

Root - 0
Floatable overlay - 0

Example:

LINKER TEXT -COUT !LPT00	Load Linker.
START TEXTEN -PT	Specify address where execution begins when root is loaded.
LINKER-300-07/08/1519	Linker identification message.
L?	Linker prompt.
IST INIT	Define INIT as the beginning of initialization code.
L?	
LINK OBJ1,OBJ2	Request that OBJ1.O and OBJ2.O be linked.
L?	
MAP	Cause OBJ1.O and OBJ2.O to be linked, and produce a link map.
L?	
OVLY ABLE	Designate end of the root, and that a nonfloatable overlay named ABLE immediately follows. The Linker assigns the number 00 to this overlay.

```
L?
BASE =OBJ2
```

Subsequent object unit(s) constituting overlay ABLE will be linked starting at the base address of the object unit OBJ2.O; this address can be determined from the map. Unprotected symbols that define locations equal to or greater than the address of OBJ2 are removed from the symbol table.

```
L?
LINK OBJ5
```

Request that OBJ5.O be linked.

```
L?
MAP
```

Request the status of symbol table.

```
L?
LINK OBJ6
```

Request that OBJ6.O be linked.

```
L?
OVLY FOX
```

Designate the end of the above overlay, and specify that a non-floatable overlay named FOX immediately follows. The Linker assigns the number 01 to this overlay.

```
L?
BASE $
```

Subsequent object unit(s) constituting the overlay named FOX will be linked starting at one location higher than the ending address of OBJ6.O. This is the default BASE address, so BASE $ need not be specified.

```
L?
LINK OBJA,OBJB
```

Request that OBJA.O and OBJB.O be linked.

```
L?
MAP
```

Request the status of the symbol table and cause OBJA.O and OBJB.O link requests to be honored, i.e., linked.

```
L?
OVLY ZEBRA
```
Designate end of above overlay 01 and name subsequent nonfloatable overlay. The Linker assigns the number 02 to this overlay.

```
L?
BASE X'1105'
```
Designate that subsequent object units constituting overlay ZEBRA will be linked starting at relative location 1105.

```
L?
LINK OBJC
```
Object unit OBJC.O will be linked starting at relative location 1105.

```
L?
LINK OBJD
```
Request that OBJD.O be linked.

```
L?
MAP
```

```
L?
FLOVLY FLOAT
```
Designate end of above overlay, and that a floatable overlay named FLOAT immediately follows. The Linker assigns the number 03 to this overlay. This overlay will be linked starting at the default base address of 0.

```
L?
LINK OBJE
```
Request that OBJE.O be linked.

```
L?
MAP

L?
QUIT

ROOT TEXT
LINK DONE
RDY:
```

Figure 16-1 illustrates use of BASE directives in a bound unit that consists of a root and overlays. This example assumes that the bound unit being created will be executed as part of task group A1, and memory pool AA will be used by this task group. Figure 16-1 also shows memory pool AA's location in memory relative to the system pool and another pool. The object units specified by the following directives are loaded into memory pool AA during execution of the bound unit.

Figure 16-2 shows the configuration of memory pool AA at different times during execution. Note that OBJ.O of the root is overlayed by overlay ABLE and that overlay FOX is partially overlayed by overlay ZEBRA. Also note that overlay FLOAT is positioned by the Loader and is not necessarily at the location shown in the diagram.

BASE

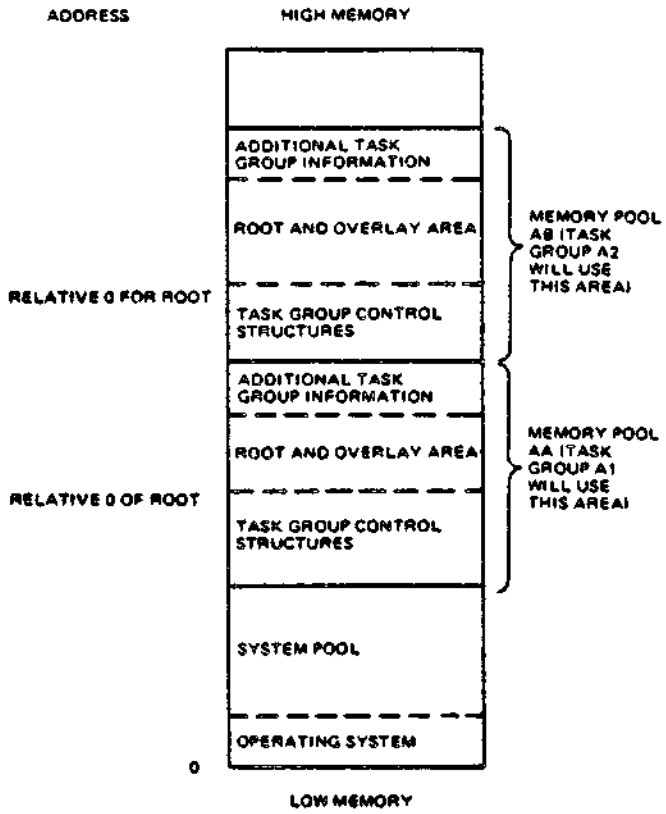


Figure 16-1. Relative Location of Memory in Memory Pool AA

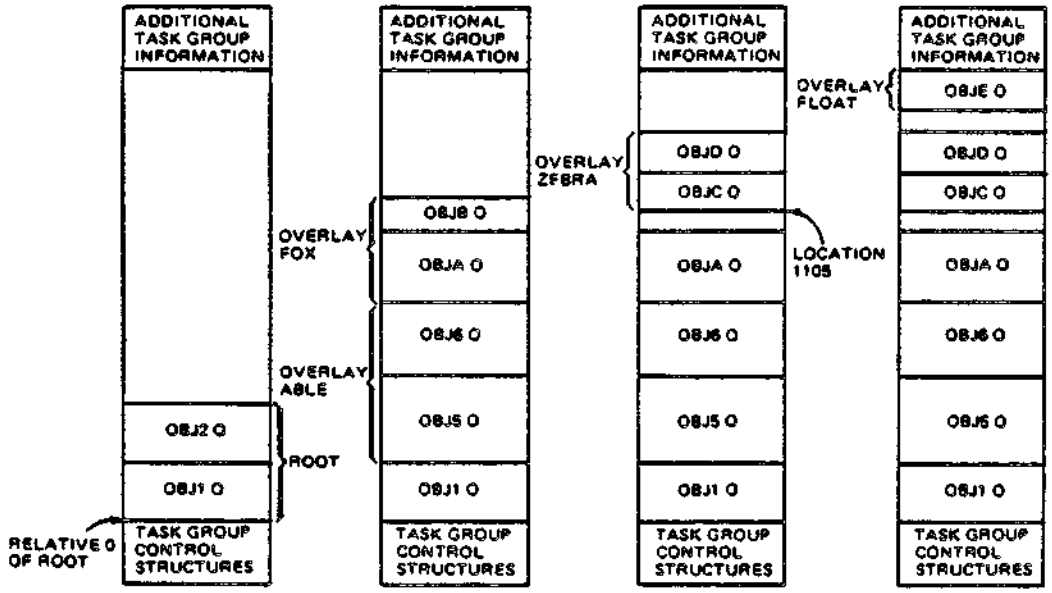


Figure 16-2. Overlays in Memory Pool AA

CC (CALL-CANCEL)

Place each overlay name and its associated Linker-generated overlay number into the bound unit attribute table so that the COBOL program can call/cancel overlays by name. This directive is used when linking COBOL programs that contain CALL/CANCEL statements that invoke overlays.

To support the CALL/CANCEL facility, two object units are required: ZCCEC.O and ZCCECO.O. ZCCEC will be automatically linked into the root, with ZCCECO linked as an overlay. These object units require only the CC link directive.

The CC directive must be specified before the first LINK, LINKN or LINKO directive in the root, and cannot be embedded in Assembly language control statements.

FORMAT:

```
CC
```

COMMON

Defines a labeled "common" area of a specified size. It may not be embedded in source code.

FORMAT:

```
{COMMON}
{COMM  }  symbol,X'size'
```

ARGUMENTS:

symbol

The external symbol to be treated as common.

X'size'

Size is specified as a one- to four-character hexadecimal number bound by single quotes and preceded by the letter X.

CPROT

Do not remove the specified common symbols from the symbol table.

This directive cannot be embedded in Assembly language control statements.

FORMAT:

```
CPROT symbol
```

ARGUMENT:

symbol

Name of the external symbol that is to be protected. The symbol must be specified in the COMM directive or defined as common during assembly or compilation.

CPURGE

Remove an unprotected common symbol from the symbol table.

FORMAT:

```
CPURGE symbol
```

ARGUMENT:

symbol

The external symbol to be removed from the symbol table. The symbol must have been defined as common.

EDEF

Make a symbolic definition available to the Loader at load time.

When EDEF is specified, the symbol's definition must already be in the Linker symbol table.

Secondary entry points of bound units, whose code is to execute under control of a task, must be defined in an EDEF directive. This includes secondary entry points of overlays and the root entry point when it will be explicitly used in a Create Group command. The start address of the root and of each overlay is placed by the Linker in the bound unit attribute table and does not need an EDEF definition. The bound unit attribute table is part of the bound unit.

If a bound unit is memory-resident, symbols (entry points and references) can be defined by EDEF so that they can be invoked by any bound unit loaded by the system. At system configuration time, when the resident bound units are loaded using the LDBU system configuration directive, these symbols are placed in the system symbol table. When the Loader loads other bound units that contain unresolved references, it tries to resolve them with the list of symbols defined for resident bound units.

If the bound unit is not memory-resident, the symbols in the attribute table of the bound unit are meaningful only as definitions of secondary entry points. Although shared bound units can be in the address space of more than one task group, the bound unit attribute table is available to the Loader only when the bound unit is being loaded. Unresolved references in any bound unit will be resolved only to symbols defined in attribute tables of resident bound units.

The EDEF directive can be embedded in Assembly language control statements.

FORMAT:

$$\left\{\begin{array}{l}\text{EDEF}\\ \text{EF}\end{array}\right\}\ \text{symbol}_1[,\text{symbol}_2]$$

ARGUMENTS:

$symbol_1$

Any external definition comprising one to six characters. The symbol must have been previously defined; it can name a root or overlay once the root or overlay has been linked. If the symbol was multiply defined, the first definition will be used.

$symbol_2$

Name of the symbol incorporated in the bound unit comprising 1 to 12 characters. If $symbol_2$ is not specified, the name of the symbol placed in the bound unit is that specified by symbol .

Example:

LINKER MYPROG -PT	Load the Linker. The bound unit named MYPROG will be created on the working directory. The list file MYPROG.M is also created on the working directory.
LINKER-300-07/08/1519	Linker identification message.
L?	Linker prompt.
LINK A	
L? LINKN B	
L? MAP	
L? EDEF B	B is a symbol previously defined by an XDEF statement in B.O as an external location or value.
L? LDEF SYM,X'1234'	Assign relative location 1234 to external symbol named SYM.
L? OVLY FIRST	Declare end of root, and name non-floatable overlay that immediately follows.

```
L?
LINK X,Y

L?
EDEF SYM

L?
QUIT                    Declare that the last Linker direc-
                        tive has been entered.  Execution of
                        the Linker terminates after the bound
                        unit has been created.

ROOT MYPROG
LINK DONE
RDY:

LINKER PROG2 -COUT !LPT00 -SIZE 2 -PT

                        Load the Linker; the bound unit to be
                        created is named PROG2.  The list file
                        is the printer.  The symbol table is a
                        maximum of 2K words of memory.

LINKER-300-07/08/1519   Linker identification message.

L?
BASE X'2222'            Subsequent object units will be loaded
                        into memory starting at the relative
                        address 2222.

L?
LINKN W                 Request that object unit W.O be
                        linked.

L?
MAP                     Produce a link map; in this map, it is
                        determined that object unit W.O
                        contains an unresolved reference to
                        the symbol SYM, which was defined in
                        the root of the bound unit MYPROG.
```

If MYPROG is loaded into memory via an LDBU configuration directive, when the Loader loads PROG2 the Loader will resolve the unresolved reference in PROG2 to the symbol SYM, which was defined in the root of MYPROG.

NOTE

An EDEF directive cannot be entered on the directive line in which the object unit is specified. For example, if the symbol TAG is defined in object unit A, the following directive line is not allowed: LINK A;EDEF TAG.

FLOATB6

Suppress certain error checking on local common references when the -R argument has not been used. The directive tells the Linker that the user will manage $B6 himself and causes each local common reference to be relocated as if the $B6 pointed to the base of the floatable or fixed overlay containing the reference. Normally, $B6 is set by the system to the base of the fixed (root and fixed overlay) area, and local common references within floating overlays would be invalid.

Before using this directive, consult with the person responsible for system building and determine available system memory.

This directive must be specified before the first object unit containing a local common reference is linked.

FORMAT:

```
FLOATB6
```

FLOVLY

Assign the specified name and a number to the floatable overlay that immediately follows, and designate the end of the preceding root or overlay. The characteristics of floatable overlays are described at the end of this directive description.

FLOVLY must be specified as the first directive of each floatable overlay.

The Linker assigns a two-digit number to each overlay. Overlays are numbered sequentially in ascending order; the first overlay is 00.

FORMAT:

FLOVLY name

ARGUMENTS:

name

Name of the floatable overlay that immediately follows. The overlay name must consist of one to six alphanumeric characters; the first character must be alphabetic.

Example:

LINKER BU -PT	Load the Linker and designate BU as the bound unit name.
LINKER-300-07/08/1519	Linker identification message.
L? LINK A,B	
L? MAP	Produce a link map.
L? FLOVLY GR	Declare the end of the root that consists of object units A.O and B.O, and specify that the next overlay is a floatable overlay named GR. The Linker assigns the number 00 to this overlay.

```
L?
LINK X,Y; MAP
L?
FLOVLY BR
```

Declare the end of floatable overlay GR and designate that the floatable overlay that immediately follows as BR. The Linker assigns the number 01 to this overlay.

```
L?
LINK R6

L?
MAP

L?
QUIT
ROOT BU
LINK DONE
```

NOTE

External location definitions defined within a floating overlay will automatically be purged at the end of the overlay, because they cannot be referenced from outside the overlay.

A floatable overlay must have the following characteristics:

1. External location definitions in the overlay are not referenced by the root or any other overlay.

2. There cannot be external references between floatable overlays.

3. The overlay does not contain external references that are not resolved by the Linker.

4. The overlay must be linked after all desired nonfloatable overlays have been linked.

5. The overlay cannot contain P+DSP references to any other overlay in the root.

6. The overlay cannot contain IMA (immediate memory address) references within itself.

7. There can be IMA references (with or without offsets) to locations in the root or any nonfloatable overlay.

GSHARE

Indicates that the bound unit is globally sharable, which means that the program is sharable between groups and the root is always loaded into the system memory pool. This directive should not be used if a SHARE directive would suffice. System performance may be affected if this directive is misused. Floatable overlays are loaded into user space and are not sharable unless overlay area tables (OATs) are used.

Before using this directive, consult with the person responsible for system building and determine available system memory.

NOTE

Nonsharable bound units (linked without SHARE or GSHARE) are always loaded into the user's memory pool.

FORMAT:

```
GSHARE
```

IN

Change the primary directory. The primary directory is the first that the Linker searches for the specified object unit(s) to be linked. The default primary directory is the working directory.

NOTE

The IN directive must be specified before the first LINK, LINKN, or LINKO directive that requests the linking of an object unit that is in the specified directory.

The specified directory remains the primary directory until another IN directive is entered. If the primary directory is changed via an IN directive and at a later time you want the task group's working directory to be the primary directory, enter the IN directive and omit the pathname.

FORMAT:

```
IN [path]
```

ARGUMENTS:

[path]

Pathname of the directory being designated as the primary directory. The pathname can contain a maximum of 57 characters. A simple, relative, or absolute pathname can be specified (methods of designating pathnames are described in Section 14 of this manual). If path is omitted, the working directory becomes the primary directory.

NOTE

The IN directive can not be embedded in Assembly language control (CTRL) statements.

Example 1:

```
    IN^DIR>PRIM
```

This directive designates that ^DIR>PRIM is the primary directory.

Example 2:

This example illustrates use of the IN directive in conjunction with directives that request the linking of object units. Assume that the primary directory is the working directory, whose relative pathname is WORK>CURR; object units X.O and Y.O, are in the working directory. A.O and C.O are not in the working directory.

LINKER OUTPUT -PT	Load the Linker; a bound unit named OUTPUT will be created on the working directory.
LINKER-300-07/08/1519	Linker identification message.
L? LINKN X	Request the linking of object unit X.O; X.O is in the working directory.
L? IN ^NEW>PRIM	Designate ^NEW>PRIM as the primary directory.
L? LINKN A,C	Request the linking of object unit A.O and C.O in the primary directory. ^NEW>PRIM>A.O is the pathname of A.O and ^NEW>PRIM>C.O is the pathname of C.O, as expanded by the Linker.
L? IN	Designate the primary directory as the working directory.
L? LINKN Y	Request the linking of object unit Y.O, in the working directory. WORK>CURR>Y.O is the pathname of Y.O, as expanded by the Linker.
L? MAP, QUIT	

INCLUDE

INCLUDE

Accept directives from a file other than user-in or the file specified in the -IN ECL argument. When the Linker encounters an end of file or a RETURN directive in the file specified by the INCLUDE directive, it again seeks directives from the previously active file. If used, the INCLUDE directive must be the last directive entered on a line.

FORMAT:

```
INCLUDE [path]
```

ARGUMENT:

[path]

Pathname of the file from which the Linker directives are to be read. A simple pathname can be up to 12 characters in length; an absolute pathname can be up to 57 characters in length.

Example:

```
INCLUDE !READER
```

This directive causes the Linker to accept directives from the card reader.

NOTES

1. The directive file specified by the INCLUDE directive cannot contain an INCLUDE directive.
2. The INCLUDE directive cannot be embedded in Assembly language control statements.

IST

Identifies the beginning of the initialization start address in the root. Initialization code is to be executed once, immediately after the root is loaded at system boot time. After the initialization code is executed, its space can be made available for overlays. The IST directive must be associated with an LDBU directive that specifies an initialization subroutine table (IST). LDBU, a CLM directive, is explained in the System Building and Administration manual. IST does not execute unless the bound unit is specified in an LDBU directive.

FORMAT:

$$\left\{\begin{matrix}\text{IST}\\ \text{IT}\end{matrix}\right\} \text{external symbol}$$

ARGUMENTS:

external symbol

Symbol specified by label in IST section of LDBU.

NOTE

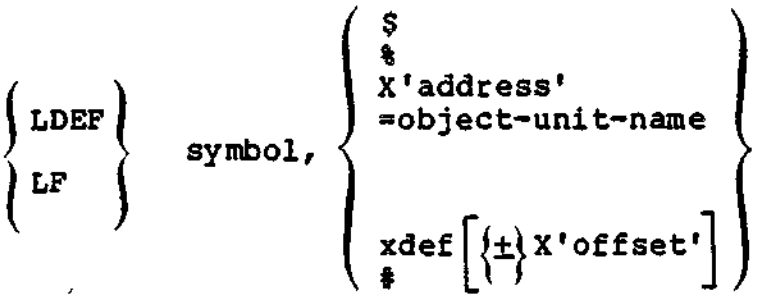
The IST directive cannot be embedded in Assembly language control statements.

LDEF

LDEF

Assign a relative location to an external symbol. A symbol should be defined only once, either as a location or as a value. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used to resolve references to the symbol during linking. When a symbol defined as a location is no longer used, its symbol table entry can be cleared by specifying the PURGE directive. PURGE has no effect if a PROTECT (PROT) directive was previously specified.

FORMAT:



ARGUMENTS:

symbol

One to six characters, each of which must be an alphanumeric character, a dollar sign ($), a period (.), or an underscore (_). The first character must be a letter or a dollar sign.

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

Hexadecimal address comprising one to five integers enclosed in single quotation marks and preceded by X. The specified address is relative to the beginning of available memory (relative 0) in the memory pool.

=object-unit-name

Specified object unit's base address.

xdef[± X'offset']

Address of any previously defined external symbol. If an offset is specified, it must be a hexadecimal integer with an absolute value less than 8000 (32768 decimal).

#

The current address.

NOTE

The LDEF directive cannot be embedded in Assembly language control (CTRL) statements.

Example:

```
LINKER BOUND -PT          Load the Linker and designate
                          BOUND as the bound unit name.

LINKER-300-07/08/1519     Linker identification message.

L?
LINK A, B, C

L?
MAP

L?
LDEF SYM, X'1234'         SYM assigned relative location 1234.

L?
OVLY FIRST                Declare end of root and name first
                          nonfloatable overlay.

L?
LINK R; MAP

L?
LDEF QUIZ,=C              QUIZ assigned base location of the
                          previously linked object unit named
                          C.O.

L?
OVLY SECOND
```

```
L?
LINKN D; LINK F; MAP

L?
LDEF NEW,SYM
```

NEW assigned same location as the symbol SYM, which was defined in the root; i.e., NEW is assigned relative location 1234.

```
L?
OVLY NEXT

L?
BASE X'1300'

L?
LINK W,X; MAP

L?
LDEF ANY,$
```

ANY assigned next location after highest address of the previously linked nonfloatable overlay, SECOND.

```
L?
OVLY THIRD

L?
LINK Z

L?
LINK Q; MAP

L?
LDEF FIND,%
```

FIND assigned next location after highest address of the root or <u>any</u> previously linked nonfloatable overlay. (A previous nonfloatable overlay was named SECOND; if it ended at location 1566 and this is the highest location reached during the linking of object units constituting this bound unit, FIND would be assigned location 1567.)

```
L?
QUIT
ROOT BOUND
LINK DONE
RDY:
```

This example illustrates the use of each format of the LDEF directive.

LIB or LIB1

Designate a directory as the secondary directory. This directive permits the linking of object units that are in directories other than the primary directory. If an object unit specified in the LINK, LINKN, or LINKO directive cannot be found in the primary directory, the Linker searches the secondary directory.

If LIB is not specified, there is no secondary directory; the Linker searches only the primary directory.

The specified secondary directory remains in effect until the LIB directive is respecified with a different directory name, or without any directory name.

All specified object units in the primary directory are linked first; then all specified object units in the secondary directory are linked, and so on. To cause object units to be linked in an order that is independent of their location, the LINKN or LINKO directive must be used.

NOTES

1. The LIB directive must be specified before the first LINK, LINKN, or LINKO directive that requests the linking of an object unit in the secondary directory.

2. This directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:

LIB [path]

ARGUMENT:

[path]

Pathname of the directory being designated as the secondary directory. A relative or absolute pathname can be specified. (Methods of specifying pathnames are described in Section 14.) If path is omitted, no search of that secondary directory is made.

Example 1:

```
LIB DIR>SECND
```

This directive designates DIR>SECND as the relative pathname of the secondary directory.

Example 2:

LIB DIR>SECND	Designate DIR>SECND as the relative pathname of the secondary directory.
LINK B	Request the linking of object unit B.O; B.O resides in the primary directory.
LINK A	Request the linking of object unit A.O; A.O resides in the primary directory.
LINK W	Request the linking of object unit W.O; W.O resides in the secondary directory. DIR>SECND>W.O is the full pathname of W.O, as expanded by the Linker.

This example illustrates usage of a secondary directory that contains unit W.O, Y.O, and Z.O.

LIB{2 3 4}

LIB {2 3 4}

Designate directories as the third, fourth, or fifth directory. If an object unit specified in the Linker directive cannot be found in the primary or secondary directory, then the third directory is searched and so on.

The specified directories remain in effect until another LIB2, LIB3, LIB4 statement is given.

NOTES

1. The LIB2, LIB3, LIB4 directive must be specified before the first LINK, LINKN, or LINKO directive that requests the linking of an object unit that is in one of these directories.

2. The LIB2, LIB3, LIB4 directive cannot be embedded in assembly language control statements.

FORMAT:

{LIB2 LIB3 LIB4} [Δpath]

ARGUMENT:

[path]

Pathname of the third, fourth, or fifth directory to be searched (if LIB is specified) if the object unit specified in a Linker directive is not found in the preceding directories. A simple, relative, or absolute pathname can be specified. If path is omitted, the specified directory (2, 3, or 4) is removed from the list of directories to be searched by the Linker.

LINK

Link one or more specified object units. Each specified object unit name is put into the link request list. The object units are linked when the first subsequent directive other than LINK or START is encountered. When this occurs, the Linker searches the primary directory and links the specified object units in the primary directory in the order that they were requested. If all of the object units are not found and there is a secondary directory, the Linker searches the secondary directory and links specified object units found there, in the order that they were requested. If there is a copy of an object unit in both the primary and secondary directory, the copy in the primary directory is linked.

The order in which object units are linked is important for the following reasons: (1) it determines which object units will be in memory when parts of the root or overlay are overlaid (2) within the root and each overlay, the first start address encountered by the Linker (either in an END statement or a START directive) is used as the start address for that root or overlay.

During each execution of the Linker, at least one LINK, LINKN, or LINKO directive must be entered for each root or overlay. Multiple LINK directives can be specified within a single root or overlay. If LINK and/or LINKN and/or LINKO directives request that the same object unit be linked more than once within a single bound unit, only the first request is honored, unless the object unit name has been purged.

LINK directives can be embedded in Assembly language control statements; the specified object unit(s) are added to the end of the current link request list. See "LINKN Directive" and "LINKO Directive" for the order in which object units are linked if there are embedded LINK directives and/or LINKN and/or LINKO directives.

FORMAT:

$$\left\{\begin{matrix}\text{LINK}\\\text{LK}\end{matrix}\right\}\ \text{obj-unit}_1\ [,\text{obj-unit}_2]\ldots$$

ARGUMENTS:

obj-unit

Name of an object unit to be linked. An object unit name consists of one to six characters, each of which must be an alphanumeric character or a dollar sign ($), a period (.), or an underscore (_). If multiple object units are specified, they are linked in the most efficient order. The first character must be a letter or a dollar sign ($).

LINKN

Link object units in the exact order specified.

If directives request that an object unit be linked more than once within a single bound unit, only the first request is honored, unless the object unit name has been purged.

During each execution of the Linker, at least one LINKN, LINK, or LINKO directive must be specified for each root or overlay.

Multiple LINKN directives can be specified within a single root or overlay.

LINKN directives can be embedded in Assembly language control (CTRL) statements; the specified object unit(s) are added to the end of the link request list and the library search restarts at the primary directory.

FORMAT:

$$\left\{\begin{matrix}\text{LINKN}\\ \text{LN}\end{matrix}\right\} \text{obj-unit}_1\,[,\text{obj-unit}_2]\ldots$$

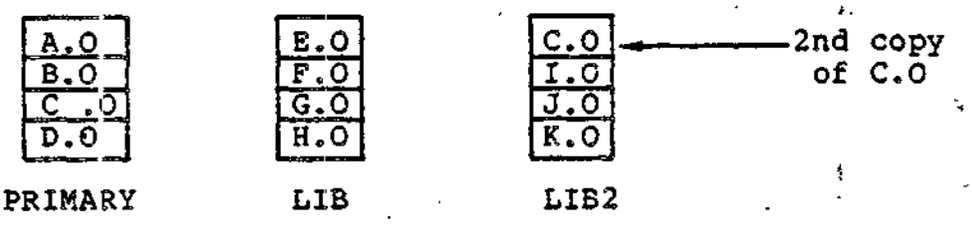
ARGUMENT:

obj-unit

Name of an object unit to be linked. An object unit name must be one to six alphanumeric characters and must not include a suffix; the first character must be a letter or dollar sign ($). The Linker appends the suffix .O to each object unit name and searches for the specified object unit name, including the suffix.

Examples of LINK and LINKN

In the following examples, assume that the working directory is the primary directory and LIB and LIB2 directives have been specified.



Example 1:

```
LINK A,G,K,C,F
```

The modules will be linked in the following order:

A,Cp,G,F,K

Example 2:

```
LINKN A,G,K,C,F
```

The modules will be linked in the following order:

A,G,K,C,F

Example 3:

```
LINK A,G,K,Cρ,F
```

Assume that module G.O contains "CTRL LINK B,J". The modules will be linked as follows:

```
A,Cρ,G,F,K,B,J
```

Once Linker has started to search LIB, it does not return to the primary directory unless a new link request list is found. The two embedded requests were added to the current link request list, forcing a rescan of all libraries.

Example 4:

```
LINKN A,G,K,C,F
```

Assume that module G.O contains "CTRL LINKN B,J". The modules will be linked as follows:

```
A,G,K,C,F,B,J
```

Example 5:

```
LINKN G,B
```

Assume that module G.O contains "CTRL LINK C". The modules will be linked as follows:

```
G,B,Cρ
```

Example 6:

```
LINK G,D,F
```

Assume that module G.O contains "CTRL LINK C,B". The modules will be linked as follows:

```
D,G,C,B,F
```

Example 7:

```
LINK G,D,F
```

Assume that module G.O contains "CTRL LINKN C,B". The modules will be linked as follows:

```
D,G,F,C,B
```

In this example, C and B are not added to the current link request list because LINKN was specified instead of LINK.

LINKnn

Link the specified object unit(s) if bit nn is turned on. This directive allows selective linking.

The LINKnn directive must be used in conjunction with the VDEF directive (or a VALDEF directive in a compilation unit). The VDEF directive is used to modify the bit setting in a 32-bit array. The leftmost 16 bits in the array are set by the symbol Z_MSKR; the rightmost 16 bits in the array are set by the symbol Z_MSKU. Through the VDEF directive, you assign a value to Z_MSKR or Z_MSKU that sets the appropriate bit "on" (a value of 1) or "off" (a value of 0).

Each occurrence of LINKnn causes the array to be indexed by nn. If the referenced bit is on (1), the link request is processed. If the referenced bit is off (0), the link request is ignored.

The bits in the array are initially set on; i.e., all LINKnn directives are processed. The array is modified by the VDEF directive (as described above). The VPURGE directive must be used to remove Z_MSKR and Z_MSKU from the symbol table before these symbols can be redefined.

FORMAT:

LINKnn obj-unit$_1$ [,obj-unit$_2$...]

ARGUMENTS:

nn

Two-digit hexadecimal value between 00 and 1F used as an index in a 32-bit array.

obj-unit$_n$

Name of the object unit to be verified for linking.

LINKO

Operate in the same manner as the LINKN directive, except that all embedded link directives in the named object units are ignored.

Only the object units named are linked.

The LINKO directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:

$$\left\{\begin{matrix}\text{LINKO}\\\text{LO}\end{matrix}\right\}\text{obj-unit}_1\ [,\text{obj-unit}_2]\ldots$$

ARGUMENT:

obj-unit

Name of an object unit to be linked. An object unit name must be one to six alphanumeric characters and must not include a suffix; the first character must be a letter or dollar sign ($). The Linker appends the suffix .O to each object unit name and searches for the specified object unit name, including the suffix.

LSR

List the Linker search rules. The directories to be searched by the Linker for object unit(s) are listed in the order in which they will be searched.

The LSR directive can be embedded in Assembly language control (CTRL) statements.

FORMAT:

```
LSR
```

MAP and MAPU

Create a link map containing: (1) defined symbols that were not purged and (2) undefined symbols to be written to the list-file (see -COUT in the Linker command).

The MAPU directive lists only undefined symbols. Both the MAP and MAPU directives can be embedded in Assembly language control statements.

If MAP is specified, each defined and undefined symbol generated by the linking of object units is listed in the map and preceded by the name of the object unit in which it is located. A map also includes the names of object units that were linked because of embedded Linker directives, and the symbols contained in those object units. If the MAP directive immediately precedes a QUIT directive, the link map will contain all the defined symbols and undefined symbols of the completed bound unit that have not been removed (i.e., purged).

If MAPU is specified, the map contains each undefined symbol and the object unit in which it is located.

MAP and MAPU directives can be interspersed among other Linker directives. When these directives are encountered all object units named in the link request list are linked before a map is produced. Maps are useful for determining whether all required object units have been linked, and whether all symbols referenced in those object units have been defined.

If there are any undefined references remaining after the last object unit is linked, a MAPU directive is automatically generated by the linker.

FORMAT:

```
{MAP }
{MP  }
{MAPU}
{MU  }
```

Default: No map produced.

A full link map (a map generated by the MAP directive) comprises the following sections:

START — Address at which execution of the root or overlay will begin; specified in the START directive or in a linked object unit.

LOW	Lowest memory address at which the current root or overlay was based.
HIGH	Next location after the highest address of the current root or overlay.
$COMM	Address assigned to COMMON for the bound unit. If no common defined, this does not appear on the MAP.
CURRENT	Next location after the current address of the root or overlay (when the map was created).
EXTERNAL DEFINITIONS	All external symbols currently defined in the symbol table. Unprotected symbols defined in the root or a previously linked overlay will appear in the map unless the symbols are purged via a PURGE or BASE directive. Symbols erroneously defined as both a value and a location will appear twice under EXT DEFS.
UNDEFINED REFERENCES	All references to undefined symbols contained in the object unit root and overlay(s) are listed in the map. For the root and each overlay containing undefined symbols, the following information is presented: • Root and overlay(s) containing references to undefined symbol(s) • Relative address of the last reference to the symbol If an undefined symbol is referenced in multiple overlays, the symbol will be listed in the map more than once. If there are external references in both P-relative and Immediate Memory Address forms to an undefined symbol, the symbol is listed twice under UNDEF.

Figure 16-3 illustrates the formats of maps generated by the MAP and MAPU directives.

NOTE

The date and time at which the bound unit was created is automatically put in the bound unit's attribute section.

```
LINKER-0300-06/18/0912                          GCOS6 MOD400-L3.0-86/18/1621
BU= EXMPL        LINKED ON: 1982/07/07 1048:04.4  -BLIC -R  -SYN

-> LIB ^CYLON>LOD>ZF1RT

-> IN ^M4LNKR

-> LIB2 ^M4LNKR>TESTPROGS>ASSEMBLER

-> VDEF Z_MSKR,X'4000'

-> BASE X'10'

-> START START1

-> LINK RTPROG

-> MAP

^M4LNKR>TESTPROGS>ASSEMBLER>RTPROG.O
RTPROG                (000010)
1981/08/31 0821:21.1 ASSEMBLER- 2.1-01/09/0913  GCOS6/MINICS 17.26.0
COMMON: COMM1    SIZE: 000032 ADDRESS: 000000
COMMON: COMM2    SIZE: 000064 ADDRESS: 000032
LOCAL COMMON: LCOMM1  SIZE: 000032 ADDRESS: 000096
LOCAL COMMON: $LCOMM  SIZE: 000064 ADDRESS: 0000C8
    C  COMM1  000000    C  COMM2  000032    C  LCOMM1 000096    C  $LCOMM 0000C8
       START1 00001E       ST2    000011       ST3    000013       ST4    000015
       ERP    000017       OLI1   000010       OLI2   000025       OLI3   00002D

[ EDEF START1]

[ EDEF OLI1]

[ EDEF OLI2]

[ EDEF OLI3]

       OLI4   000035
```

LINKER ID, LINK DATE, AND LINE

LINKER DIRECTIVES

COMMON BLOCKS DEFINED IN RTPROG

EMBEDDED LINKER DIRECTIVES

Figure 16-3. Link Map Formats

```
* * * * * * * * * * * * * * * * * * *
* * * * *       M A P       * * * * *
* * * * * * * * * * * * * * * * * * *

EXMPL                1982/07/07 1048:04.4

**START:      00001E
**LOW:        000010
**HIGH:       00003F
**CURRENT:    00003F


***** COMMON BLOCK DEFINITIONS *****

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000    C  COMM2  000032    CX LCOMM1 000096    CX $LCOMW 0000C8


***** EXTERNAL DEFINITIONS *****

     P ZHCOMM 000000     P ZHREL  000000      Z_MSKR   4000

** EXMPL  000010

 * RTPROG 000010
    C  COMM1  000000    C  COMM2  000032    CX LCOMM1 000096    CX $LCOMW 0000C8
       START1 00001E       ST2    000011       ST3    000013       ST4    000015
       ERR    000017       OLI1   000010       OLI2   000025       OLI3   00002D
       OLI4   000035


***** UNDEFINED REFERENCES *****

** EXMPL  000010
 * RTPROG 000010
     V OVLAY1 00001F     V OVLAY2 000029     V OVLAY3 000033     V OVLAY4 000039
```

Figure 16-3 (cont). Link Map Formats

```
KEY: *=OBJECT FILE NAME, **=ROOT OR OVERLAY NAME, ON HEADING;  C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *

-> OVLY OVLAY1                          LINKER DIRECTIVES

-> LINKN RTPROG.00

^M4LNKR>TESTPROGS>ASSEMBLER>RTPROG.00.0
OVLAY1                  (00003F)
1979/08/08 1408:00.6 ASSEMBLER-0200-07/16/0800  GCOS6 MOD400-L200-07/20/2251
COMMON: COMM1    SIZE: 000032 ADDRESS: 000000
COMMON: COMM2    SIZE: 000064 ADDRESS: 000032
LOCAL COMMON: LCOMM1  SIZE: 000032 ADDRESS: 00012C
LOCAL COMMON: $LCOMM  SIZE: 000096 ADDRESS: 00015E
LOCAL COMMON: LCOMM2  SIZE: 000032 ADDRESS: 0001F4
    C  COMM1  000000   C  COMM2  000032   C  LCOMM1 00012C   C  $LCOMM 00015E
    C  LCOMM2 0001F4                                                          COMMON BLOCKS DEFINED IN OVLAY1

-> LINK01 TIME

-> LINK02 WAIT                          LINK DIRECTIVES

-> LDEF ST1A,ST4+X'10'

^CYLON>LDD>ZF1RT>TIME.O
TIME    82020100   (000053)
1982/03/01 1331:23.8 ASSEMBLER- 2.1-01/09/0808  GCOS6 MOD400-L3.0-01/19/0944

[ LINK Z1TIME]

^CYLON>LDD>ZF1RT>Z1TIME.O
Z1TIME  82020100   (000053)
1982/02/19 0903:38.5 MAP-1.0 -10/25/81    GCOS6 MOD400-L3.0-01/19/0944   PAGE
       TIME   000053
```

Figure 16-3 (cont). Link Map Formats

```
=> EDEF ST3

=> MAP


* * * * * * * * * * * * * * * * * *
* * * * *      M A P      * * * * *
* * * * * * * * * * * * * * * * * *

EXMPL                1982/07/07 1048:04.4

**START:      000045
**LOW:        00003F
**HIGH:       000078
**CURRENT:    000078

***** COMMON BLOCK DEFINITIONS *****

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000   C  COMM2  000032   CX LCOMM1 000096   CX $LCOMM 0000C8

** OVLAY1 00003F
 * .00    00003F
    CX LCOMM1 00012C   CX $LCOMM 00015E   CX LCOMM2 0001F4


***** EXTERNAL DEFINITIONS *****

     P ZHCOMM 000000    P ZHREL  000000      Z_MSKR   4000

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000   C  COMM2  000032   CX LCOMM1 000096   CX $LCOMM 0000C8
       START1 00001E      ST2    000011      ST3    000013      ST4    000015
```

ADDRESS INFORMATION FOR OVLAY1

ADDRESS INFORMATION CONTINUED

Figure 16-3 (cont). Link Map Formats

```
       ERP    000017       OLI1   000010       OLI2   000025       OLI3   000020
       OLI4   000035       OVLAY1   0001

** OVLAY1 00003F
 * .00    00003F
    CX LCOMM1 00012C    CX SLCOMM 00015E    CX LCOMM2 0001F4
 * TIME   000053
 * ZITIME 000053
       TIME   000053       ST4A   000025


****** UNDEFINED REFERENCES *****

** EXMPL  000010
 * RTPROG 000010
     V OVLAY2 000029     V OVLAY3 000031     V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING;  C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * *

-> MAPU                                          MAPU


* * * * * * * * * * * * * * * * *
* * * * *     M A P     * * * * *
* * * * * * * * * * * * * * * * *

EXMPL              1982/07/07 1048:04.4

**START:     000045
**LOW:       00003F
**HIGH:      000078
```

Figure 16-3 (cont). Link Map Formats

```
**CURRENT:  000078


***** UNDEFINED REFERENCES *****

** EXMPL  000010
 * RTPROG 000010
     V OVLAY2 000029     V OVLAY3 000031     V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING;  C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *

-> PROTECT OL14

-> PURGE ST2

-> BASE OL13

-> OVLY Ov_AY2

-> LINK0 RTPROG.01

^MALNKP>TESTPROGS>ASSEMBLER>RTPROG.01.0
OVLAY2              (000020)
1981/05/31 0921:29.4 ASSEMBLER- 2.1-01/09/0913  GC056/MINICS 17.26.0
COMMON: COMM1    SIZE: 000032 ADDRESS: 000000
COMMON: COMM2    SIZE: 000064 ADDRESS: 000032
LOCAL COMMON: LCOMM1  SIZE: 000032 ADDRESS: 00022e
LOCAL COMMON: SLCOMM  SIZE: 0000CA ADDRESS: 000258
LOCAL COMMON: LCOMM2  SIZE: 000032 ADDRESS: 000320
    C  COMM1  000000   C  COMM2  000032   C  LCOMM1 00022e   C  SLCOMM 000258
    C  LCOMM2 000320
```

LINKER DIRECTIVES

COMMON BLOCKS DEFINED IN OVLAY2

Figure 16-3 (cont). Link Map Formats

```
-> LINK01 WAIT
                    ] EMBEDDED LINK
-> MAP

^CYLON>LDD>ZF1RT>WAIT.O
WAIT    82020100   (000041)
1982/03/01 1331:37.0 ASSEMBLER- 2.1-01/09/0808  GCOS6 MOD400-L3.0-01/18/0944

: LINK Z1WAIT;

^CYLON>LDD>ZF1RT>Z1WAIT.O
Z1WAIT  82020100   (000041)
1982/02/19 0909:24.5 MAP-1.0 -10/25/81    GCOS6 MOD400-L3.0-01/18/0944   PAGE
       WAIT   000041


* * * * * * * * * * * * * * * * * *
* * * * *     M A P     * * * * * *   MAP #3
* * * * * * * * * * * * * * * * * *

EXMPL              1982/07/07 1048:04.4

**START:    000033



**LOW:      00002D
**HIGH:     00005A
**CURRENT:  00005A

***** COMMON BLOCK DEFINITIONS *****

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000   C  COMM2  000032
```

Figure 16-3 (cont). Link Map Formats

```
** OVLAY2 00002D
 * .01    00002D
    CX LCOMM1 000226   CX SLCOMM 000258   CX LCOMM2 000320


***** EXTERNAL DEFINITIONS *****

     P ZHCOMM 000000    P ZHREL  000000      Z_MSKR   4000

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000   C  COMM2  000032      START1 00001E      ST3    000013
       ST4    000015      ERP    000017      OLI1   000010      OLI2   000025
     P OLI3   000035      OVLAY1   0001

** OVLAY1 00003F
 * .00    00003F
 * TIME   000053
 * Z1TIME 000053
       ST44   000025      OVLAY2   0002

** OVLAY2 00002D
 * .01    00002D
    CX LCOMM1 000226   CX SLCOMM 000258   CX LCOMM2 000320
 * WAIT   000041
 * Z1WAIT 000041
       WAIT   000041


***** UNDEFINED REFERENCES *****

** EXMPL  000010
 * RTPROG 000010
     V OVLAY3 000031    V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING;  C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED
```

Figure 16-3 (cont). Link Map Formats

```
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *

-> LSR

**********
PRIM  ^M4LVKR
LIB   ^CYLON>LOD>ZF1RT
LIB2  ^M4LNKR>TESTPROGS>ASSEMBLER
**********

-> Q


* * * * * * * * * * * * * * * * * *

* * * * *     M A P     * * * * *
* * * * * * * * * * * * * * * * * *

EXMPL              1982/07/07 1048:04.4

**START:     000033
**LOW:       00002D
**HIGH:      000000
**CURRENT:   00005A


***** UNDEFINED REFERENCES *****

** EXMPL  000010
 * RTPROG 000010
     V OVLAY3 000031    V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING;  C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
```

ACTIVE OBJECT UNIT DIRECTORIES

Figure 16-3 (cont). **Link Map Formats**

```
**********
ROOT  EXMPL
HIGHEST OVERLAY NUMBER: 3
NUMBER OF EDEFS:  5
 SLIC
**********
CMMN  DATA                BASE:  000000  START:  000000  .F..  HIGH: 000352
ROOT  EXMPL               BASE:  000010  START:  00001E  ..UI  HIGH: 00003F
OVLY  OVLAY1      # 0001  BASE:  00003F  START:  000045  ..U.  HIGH: 000078
OVLY  OVLAY2      # 0002  BASE:  00002D  START:  000033  ..U.  HIGH: 00005A

KEY: S=SHAREABLE; F=FLOATING; I=CONTAINS AN IMA; U=CONTAINED AN UNDEFINED
     REFERENCE; ->=IN-LINE DIRECTIVE; [...]=EMBEDDED DIRECTIVE

**********
SIZE OF ROOT AND FIXED OVERLAYS:  000078
LAST BU RECORD NUMBER:  12

*** BU CONTAINS UNRESOLVED REFERENCES.

**********
LINK DONE
**********

*** THERE WERE 1 ERRORS DURING THE LINK.


EOF
```

Figure 16-3 (cont). Link Map Formats

OVERLAYTABLE

Include the name of each overlay and its associated Linker-generated overlay number in the set of symbols passed to the Loader at load time.

FORMAT:

```
{ OVERLAYTABLE }
{ OE           }
{ OT           }
```

OVLY

Assigns the specified name to the non-floatable overlay that immediately follows, and designates the end of the preceding root or overlay.

OVLY must be specified as the first directive of each non-floatable overlay.

The Linker assigns a two-digit number to each overlay. Overlays are numbered sequentially, in ascending order; the first overlay is 00.

FORMAT:

```
OVLY name
```

ARGUMENT:

name

Name of the nonfloatable overlay that immediately follows. The overlay name must consist of one to six alphanumeric characters; the first character must be alphabetic.

Example:

```
LINKER BU -PT                 Load the Linker and designate BU
                              as the bound unit name.

LINKER-300-07/08/1519         Linker identification message.

L?
LINK A, B; MAP

L?
OVLY A2                       Declare the end of the root (which
                              comprises object units A.O and B.O)
                              and specify that the next overlay is
                              a nonfloatable overlay named A2.
                              The Linker assigns the number 00 to
                              this overlay.

L?
LINK X

L?
LINK Y
```

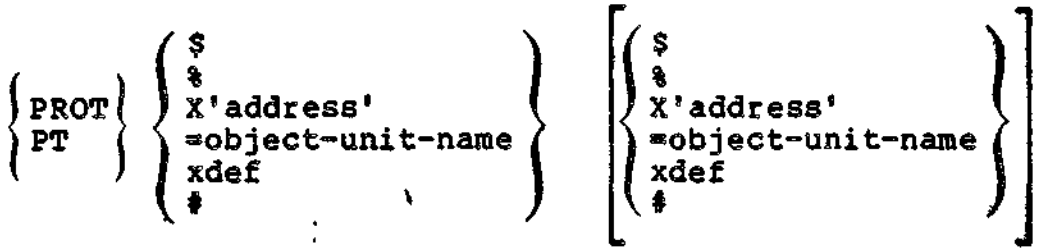
```
L
MAP

L?
QUIT
ROOT BU
LINK DONE
RDY:
```

PROTECT

Prevents certain symbols and/or object unit names from being removed from the symbol table. Symbols that identify addresses within the range of addresses specified by the first operand through the second operand are protected, and object unit names equated to addresses within that range are protected. If a second operand is not specified, the symbol at the address of the first operand and any other symbols or object unit names equated to that address are protected. Once a symbol or object unit name is protected, it cannot be purged later. The protect directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:



ARGUMENTS:

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

Hexadecimal address comprising one to five integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of the root (relative 0).

=object-unit-name

Specified object unit's base address.

xdef

Address of any previously defined external symbol.

*

The current address.

Example 1:

```
PROT X'1234',X'4565'
```

This directive protects symbols and object unit names that identify addresses from 1234 through 4565.

Example 2:

```
PT =FIRST
```

This directive protects symbols that identify the base address of the object unit FIRST and all symbols equated to that address. The base address of FIRST is determined by producing a link map (see "MAP and MAPU Directives").

Example 3:

```
PROT SYM,X'5555'
```

This directive protects symbols that identify addresses from the address of the previously defined external symbol named SYM through 5555; object unit names equated to those addresses also are protected.

PURGE

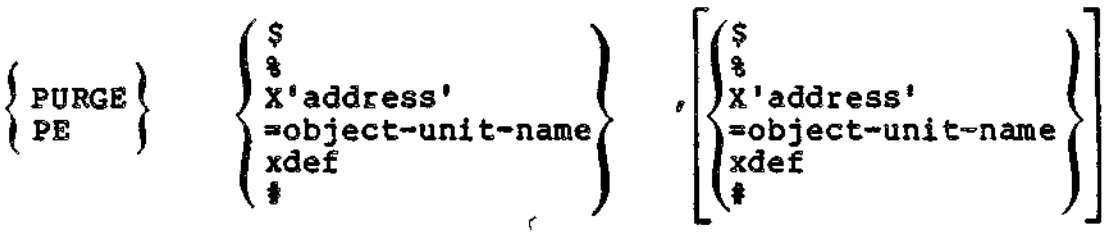
Remove the following items from the symbol table: unprotected symbols that define addresses greater than or equal to the first address and less than or equal to the second address. If a second operand is not specified, the symbol at the address of the first operand and any other symbols or object unit names equated to that address are purged.

An object unit currently being linked can contain definitions used for previously linked object units that will not be used for subsequent object units to be linked. By removing symbols that are no longer required, there is more room for symbols that will be required by subsequently linked object units.

NOTES

1. Undefined symbols cannot be purged.
2. Symbols and object unit names that are protected by a PROTECT directive cannot be purged.
3. Only symbol addresses (not values) can be purged by this directive.
4. The PURGE directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:



ARGUMENTS:

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

Hexadecimal address comprising one to five integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of the root (relative 0).

=object-unit-name

Specified object unit's base address.

xdef

Address of any previously defined external symbol.

#

The current address.

Example 1:

```
PURGE X'1234',X'4565'
```

This directive purges unprotected symbols that identify addresses from 1234 through 4565, and unprotected object unit names equated to addresses within that range.

Example 2:

```
PE =FIRST
```

This directive purges unprotected symbols that identify the base address of the load unit FIRST and any other unprotected symbol names equated to that address.

Example 3:

```
PURGE SYM,X'5555'
```

This directive purges unprotected symbols that identify addresses from the address of the previously defined external symbol SYM through 5555; unprotected object unit names equated to addresses within that range also are purged.

QUIT

Indicates that the last Linker directive has been entered. QUIT should be entered after the last overlay, or at the end of the root if there are no overlays.

If object units were successfully linked, the bound unit is completed and the Linker terminates; otherwise, the Linker terminates execution immediately.

The QUIT directive is required; it cannot be embedded in Assembly language control statements.

FORMAT:

$$\left\{\begin{matrix}\text{QUIT}\\ \text{QT}\\ \text{Q}\end{matrix}\right\}$$

RERUN RELOCATABLE (RR)

Reload the sharable bound unit at restart into locations other than those it occupied when the checkpoint was taken (see the Commands manual for details on checkpoint-restart). If this directive is not specified, the bound unit is reloaded at the same system memory pool locations it occupied when the checkpoint was taken.

The RR directive can be embedded in Assembly language control statements.

FORMAT:

```
RR
```

NOTE

If the RR directive is used, it is important to remember that after reloading, the current values of the IMAs referencing locations in the bound unit are no longer valid; therefore, if the bound unit contains IMAs (see the link map or compiler list file to determine this), RR should not be used.

RETURN

Accept directives from the user-in file. This directive should only be specified in an INCLUDE file. A RETURN directive in a file specified in an INCLUDE directive is logically equivalent to an EOF mark; it returns the Linker to the user-in file.

FORMAT:

```
RETURN
```

SEG

Cause the bound unit to occupy one or two physical segments in memory. Before using this directive, consult with the person responsible for system building and determine the segment numbers available to task groups. The SEG directive can be entered at any point. You can specify the physical segment number(s) to be assigned as well as the access (read, write, and execute) to the segment(s). This directive is only meaningful when the bound unit is executed in a swap pool.

FORMAT:

```
{SEG}  {X}'code_segment_no' [ {,code_access[,data_access]} ]
{SG }  {Z}                   [ {,,data_access            } ]
```

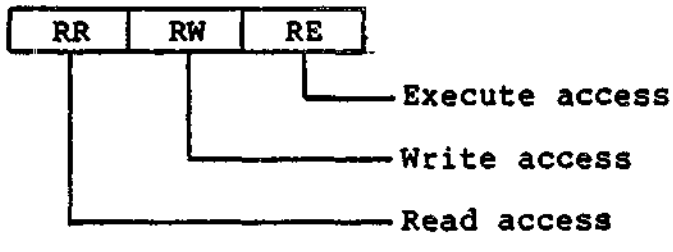
ARGUMENTS:

code_segment_no

Hexadecimal number from 1 to F that specifies the number of the physical segment containing the bound unit if the -R ECL parameter in the Linker command is not specified. When -R ECL argument is specified, code_segment_no is a hexadecimal number from 2 to F and data_segment_no is equal to code_segment_no -1. There are 15 big segments in memory. Each big segment is at most 64K. The user can use big segment numbers from 1 to F.

code_access and data_access

Bit strings of exactly 6 bits that specify the access right for the readable and writable segments, respectively. Each bit string represents the corresponding access fields in the segment descriptor. Representation of the format of the access argument is:



The two bit positions that designate each access represent ring number:

Bit	Ring Number	
11	0	Executive
10	1	Privileged real time
01	2	Unprivileged real time
00	3	Batch

A program making a reference (read, write, and execute) to memory is given access, if the value of the ring number of the program's privilege is less than or equal to ring value of the desired memory location.

The defaults for the access fields are:

Data access - -R - always 000000

Code access - sharable
- globally sharable
] 001100

- otherwise 000000

Example 1:

```
SEG X'06',,000100
```

In this example if -R was not specified, segment 6 with default access is assigned to the bound unit and the specified data access is ignored. If -R was specified, segment 6 with default access is assigned to the code and segment 5 with the specified access is assigned to the data.

Example 2:

```
SEG ,001000
```

In this example, if -R was not specified, the loader assigns a segment number with the specified access to the bound unit. If -R was specified, the loader assigns a segment number with the specified access to the code and segment number -1 with default access to the data.

SHARE

Designates a bound unit as sharable within a memory pool. If another task requests that the bound unit be loaded, instead of another copy of the bound unit being loaded, the existing copy in memory is used. The bound unit must have reentrant code, but the system does not check to see that it does.

SHARE must be specified in the definition of the root before the first overlay is defined.

SHARE directives can be embedded in Assembly language control statements.

FORMAT:

```
{SHARE}
{SE   }
```

STACK

Specifies the size of the stack in a decimal number of words. If no STACK directive is specified, the Linker will use the largest stack size specified in a object unit linked into the bound unit.

FORMAT:

```
STACK value-n
```

ARGUMENT:

value-n

The size of the stack in a decimal number of words.

START

Specifies the relative location within a root or overlay at which execution of the root or overlay will begin once it is loaded into memory by the Loader.

If a linked object unit contains a start address (an Assembler or compiler END statement was specified) and the START directive is specified, the first start address encountered (in either a START directive or an END statement) is used by the Linker for that root or overlay.

FORMAT:

```
{START}
{ST   }   symbol
```

ARGUMENT:

symbol

Name of the external symbol whose address indicates the relative address at which the root or overlay will begin executing.

Default: Start address specified in the first linked object unit that has a start address. If the symbol is never defined or a start address is not found, the start address is the first non-common location in the root or overlay.

NOTE

For very large programs, the start address must be within 64K of the beginning of the root or overlay.

SYS

Indicates that this overlay or root can be run as a system task. This directive does not control where the bound unit is loaded, rather it allows a bound unit to be executed either as a system or user task. Before using this directive, consult with the person responsible for system building and determine available system memory. The SYS directive can be embedded in Assembly language (CTRL) statements.

FORMAT:

```
{SYS}
{SS }
```

Example:

```
SYS
```

VAL

Defines a value at link time that is equivalent to the difference between two external location definitions.

VAL directives can be embedded in Assembly language control statements.

FORMAT:

$$\left\{\begin{matrix}\text{VAL}\\\text{VL}\end{matrix}\right\} \text{ symbol, external location } - \text{ external location}$$

ARGUMENTS:

symbol

Assign a name to the value of the distance between two locations.

external location$_n$

Location defined externally.

VDEF

Assigns a value to an external symbol. The VDEF directive cannot be embedded in Assembly language control statements. A symbol should be defined only once, as a value or as a location. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used during linking to resolve external references.

FORMAT:

```
{VDEF}
{VF  }   symbol,X'value'
```

ARGUMENTS:

symbol

One to six alphanumeric characters.

X'value'

Value of the designated symbol; must be a one-word hexadecimal integer enclosed in single quotes and preceded by X.

VPURGE

VPURGE

Remove the specified external value definition. This directive cannot be embedded in Assembly language control statements.

FORMAT:

```
VPURGE value-definition-symbol
```

ARGUMENT:

value-definition-symbol

External symbol name associated with a particular value.

LINKER PROCEDURES

This subsection describes the frequently used Linker procedures. The examples provided show different methods for linking COBOL programs, including one example that uses overlays.

Overview

The Linker is a system software program that functions as the final stage of program development before program execution is possible. Before linking, a program must be compiled (or assembled) to produce one or more object units or compile units that the Linker identifies for linking. The Linker recognizes object units by the .O suffix (appended to each file name by the compiler). The Linker combines one or more object units to produce a bound unit. A bound unit is an executable program consisting of a root segment and zero or more overlay segments that can be loaded into memory.

Using Overlays

In situations where memory is limited, it may be necessary for you to divide your program into one or more overlay segments so that individual portions of your program may be called into a single memory area only when they are needed. Unlike the root segment, which cannot be reloaded once it is read into memory, an overlay segment can be read in as often as it is needed. See Example 4 for a link session that uses overlays.

Interrupting Linker Execution

If at any time during Linker execution you want to interrupt processing, you can perform one of the following actions:

- Press the QUIT, INTERRUPT, or BREAK key at your terminal.
- Enter ΔCΔBID at the operator terminal, where id is your two-character task group identification.

After performing one of the above actions, a **BREAK** message will appear on your terminal. You can now:

- Enter any valid ECL command.
- Resume Linker execution as if no break had occurred by entering the Start (SR) command.

- Terminate Linker processing and return to command level by entering the Unwind (UW) command.

- Restart your task group by issuing a New Process (NEW_PROC) command.

NOTE

If you want to terminate the MAP operation and jump to the next Linker directive, issue a Program Interrupt (PI) command.

Section 17
SINGLE-USER DEBUGGER

This section describes the programming debugging facility $D DEBUG.

OVERVIEW

$D DEBUG is an interactive testing and error correction facility used at program execution time to debug bound units.

$D DEBUG runs in the dedicated task group $D. Any breakpoints you set can be encountered by all users. (Breakpoints are discussed later in this section.) Therefore, you should ensure that you are the only user on the system when utilizing the $D DEBUG. Note that you must use the $D DEBUG facility when you are debugging the lead task or a sharable bound unit.

$D DEBUG CAPABILITIES

Data, referred to by program locations in terms of memory addresses, are displayed in hexadecimal or ASCII dump format. Using $D DEBUG directives, you can suspend programs at selected breakpoints during execution and examine input, display data, or alter values.

With the $D DEBUG facility you can:

- Define, store, and execute a sequence of directives
- Set or clear true breakpoints in task code to monitor task status
- Set or clear bound unit breakpoints to gain control of bound units as they are loaded
- Display, change, and dump either memory or registers
- Evaluate expressions.

- Do not enter a semicolon after the last directive on a line. Press carriage return after the last (or only) directive on a line.

- Enter all argument values in hexadecimal notation, except where specified otherwise.

Special symbols are used in $D DEBUG directive lines. These symbols are described in Table 17-1.

Table 17-1. Symbols Used in $D DEBUG Directive Lines

Symbol Type	Meaning
Arithmetic Operators:	
plus sign (+)	Performs addition.
minus sign (-)	Performs subtraction.
K	Multiplies a hexadecimal integer by 1024 decimal (400 hexadecimal) when K is the last character of an integer expression.
Address Operators:	
period (.)	Represents the last start address used in a previous memory reference directive (DH, CH, DP).
ampersand (&)	Represents the address of the next location beyond the last one used by a previous memory reference directive (DH, CH, DP).
brackets [ ]	Signifies the contents of the location defined by the expression within the brackets. Three levels of nesting may be used.
Reserved Symbols:	
$Bn	Contents of base register n of the active level. The values 1 through 7 can be used for n.
$Rn	Contents of the data register n of the active level. The values 1 through 7 can be used for n.
$P	Contents of the program counter of the active level.
$I	Contents of the indicator register of the active level.
$IV	Address of the Task Control Block (TCB) of a trapped task which is currently at the head of the active level's trap queue.
$IV_Bn	Represents the contents of the base register n as stored in the Interrupt Save Area (ISA) of the active level. The values 1 through 7 can be used for n.

OIM default task group, enter the following command at any time:

```
C :$D:
```

Example 3:

Loading $D DEBUG at a user terminal, not the operator terminal:

```
SG $D GRANT.TECH 7 !KSR01 -EFN DEBUGDB -POOL AB -WD >WORK
```

$D DEBUG Operation With MMU

The $D DEBUG task group is loaded in ring 0, a privileged state, in order to run effectively in a protected (MMU) system. $D DEBUG will handle traps to trap vector 14 (unauthorized reference to protected memory) and to trap vector 15 (reference to unavailable resource) and continue as described below.

An error message will be displayed if you try to access nonvirtual memory within any $D DEBUG directive, except the Dump Memory (DP) directive. If a trap to trap vector 15 occurs when a DP directive is specified, $D DEBUG will dump as much of the requested memory as possible. Once a nonvirtual address is accessed, the rest of the current line to be printed will be blank-filled. The current nonvirtual address will be advanced to the value that is the next multiple of 1K. This procedure will continue until the area to be dumped is exhausted or the end of memory is reached.

$D DEBUG FILE REQUIREMENTS

Directive lines stored for later execution reside in a preallocated disk file DEBUG.WORK. This file must be in the volume major directory of the bootstrap device specified in the Set File (SF) directive (described later in this section). The size of the file must be 54 sectors whether on diskette, cartridge disk, or any other media.

ENTERING $D DEBUG DIRECTIVES

$D DEBUG directives consist of only a directive name and one or more arguments. When entering directives, follow the rules listed below:

- Separate arguments within each directive by at least one space.
- If you enter more than one directive on a line, separate each directive and its arguments from the next by a semicolon (;).

Table 17-1 (cont). Symbols Used in $D DEBUG Directive Lines

Symbol Type	Meaning
Debug Language (cont):	
exp	Indicates a valid expression formed by using expression elements. Expression elements are addresses, reserved symbols, and hexadecimal values up to 32 bits in length. No more than one address is allowed within an expression. An expression element may be preceded by the positive (+) or negative (-) unary operator. Expression elements can be joined by the addition (+) or subtraction (-) operator.
rexp	Consists of exp_1/exp_2, where exp_1 is a hexadecimal number that is a value of a location expression; exp_2 is an optional hexadecimal repeat factor whose value must be between 1 and 32,767. If exp_2 is omitted, there is no repetition.
;	Separation character between directives on the same line.
*	Signifies "all" in certain print, clear, and list directives.

Table 17-2 summarizes $D DEBUG directives by function. These directives are described in detail alphabetically on the following pages. In each directive's format, it is assumed that $D DEBUG was previously designated as the OIM default task group when the operator terminal is specified as user-in, so $D is not specified before each directive name.

NOTES

1. Pay careful attention to the format of each directive, because the usage of delimiters, if any, between a directive name and the first (or only) parameter varies according to which directive is being specified.

2. If a directive has a parameter in which you may specify the logical resource number (lrn) of the device on which information will be printed, $D DEBUG uses the specified device without first determining whether the device has been reserved for exclusive use by another task.

Table 17-1 (cont). Symbols Used in $D DEBUG Directive Lines

Symbol Type	Meaning
Reserved Symbols (cont):	
$IV_Rn	Represents the contents of the data register n as stored in the Interrupt Save Area (ISA) of the active level. The values 1 through 7 can be used for n.
$S	Contents of the system status register (level number and privilege bit only) of the active level.
$SL	Represents the value of the level number of the active level.
$E	Used with set bound unit breakpoint directive to represent the entry point as defined in the bound unit or by the caller. Used in place of $P associated with true breakpoints.
G through Z	Twenty single-character symbols having initial values of zero. Values may be assigned using the AS directive.
Debug Language:	
{^} < {^} = {^} >	The condition to be satisfied in an IF directive for continuous processing of the directive line. ^ indicates a logical 'NOT' which may optionally be used.
parentheses ( )	Indicate directive or header information to be stored for later use. Unmatched right parentheses result in an error. A right parenthesis that is paired with the first left parenthesis terminates the directive definition.

Table 17-2 (cont). Summary of $D DEBUG Directives by Function

Function	Directive	Directive Name
General execution	FO	Redirect output
	Hn	Print header line
	IF	Conditional execution
	LL	Specify line length of operator terminal or another terminal currently in use
	RF	Reset file location
	SF	Specify file location
	QT	Abort $D DEBUG task group

NOTES

1. The memory and register control directives (AR, AS, CH, DH, and DP) apply to registers on the active level. To determine which level is the active level and/or to set the active level to a specified value, see "Determining/ Setting the Active Level" below.

2. The following directives are predefined or delayed-execution directives and directive lines associated with them are stored in the file DEBUG.WORK: Sn, Hn, Dn, DT, SBn.

Planning Considerations

SETTING TRUE BREAKPOINTS

True breakpoints can be set to trap at selected task code locations. At true breakpoints, memory and register values can be displayed and changed. In this way, a task can be executed, the values of its variables checked as execution proceeds, code modified, and if necessary, variable values changed in order to test the sequence of code up to the next breakpoint.

The following are guidelines for setting true breakpoints:

1. True breakpoints can be set in a task group (or in an overlay in a task group) only when the task group/overlay currently is memory resident. The Set Bound Unit Breakpoint (SBn) directive should be used to gain control of a task group bound unit/overlay when it is loaded, to allow true breakpoints to be properly set.

2. True breakpoints may <u>not</u> be set in code that will be executed at the inhibit level.

Table 17-2. Summary of $D DEBUG Directives by Function

Function	Directive	Directive Name
Directive line definition and handling	Dn	Define directive line n
	En	Execute directive line n
	P*	Print all predefined directive lines
	Pn	Print directive line n
True breakpoint control	C*	Clear all true breakpoints
	Cn	Clear true breakpoint n
	GO	Proceed from breakpoint
	L*	List all true breakpoints
	Ln	List true breakpoint n and associated directive line
	Sn	Set true breakpoint n
Bound unit breakpoint control	CB*	Clear all bound unit breakpoints
	CBn	Clear bound unit breakpoint n
	LB*	List all bound unit breakpoints
	LBn	List bound unit breakpoint n
	SBn	Set bound unit breakpoint n
Trace trap control	DT	Define trace directive line
	PT	Print trace directive line
	ST	Start j-mode trace
	ET	End j-mode trace
Active level control	SL	Set active level
	TL	Set temporary active level
Memory and register control	AR	Print contents of all registers of the active level
	CH	Change memory
	DH	Display memory in hexadecimal
	DP	Dump memory in hexadecimal and ASCII
Symbol control	AS	Assign a hexadecimal value to symbol or register
	VH	Print value of expression in hexadecimal

2. The Set Temporary Level (TL) directive designates a level as the temporary active level; this permits you to display or alter registers of a level different from the default terminal level without permanently changing the default terminal level.

3. Whenever a break or trace point is processed for a task, the active level is set to the level of that task for the duration of any stored-directive line execution. After this duration, the last operator-specified value of "active level", if any, is again in force.

MAINTAINING A TRACE HISTORY

When using $D DEBUG with disk-stored directive lines that execute upon encountering a trap or a breakpoint, a trace history may be maintained on a line printer.

Also, while at a $D DEBUG true breakpoint stall, a particular task may be set to run in jump-trace mode. In this case, every departure from the current sequence of instructions generates a trace trap.

$D DEBUG DIRECTIVES

This subsection provides a detailed description of the $D DEBUG directives in alphabetical order by directive name.

The following notational symbols are used to describe the format of $D DEBUG directives.

Notational Symbols	Meaning
Braces { }	For a single enclosed argument, indicates that the argument is optional. If more than one argument is enclosed by braces in a vertical listing, the braces indicate that a choice is to be made. In this case, optional arguments are identified in the text.
Ellipsis (...)	Indicates the ability to repeat within braces.
Delta (Δ)	Indicates one or more spaces.
Vertical bar (|)	Indicates a choice between two or more arguments.

Note that the use of braces shown above differs from the usage defined in the preface.

3. If sharable code contains breakpoints, each task that uses the code encounters the breakpoint, regardless of which task group the task is in.

True breakpoints are set in tasks by specifying the Set True Breakpoint (Sn) directive; the detailed description of Set True Breakpoint directive includes additional rules for specifying true breakpoints.

CONTROLLING OUTPUT USING A BREAKPOINT

Output can be redirected from an operator terminal by using a breakpoint. When the breakpoint condition occurs, the File Out (FO) directive can be used to redirect the $D DEBUG output.

In the discussions that follow, the terminology "current $D DEBUG output device" refers to either an interactive terminal specified for a task group or the device defined by a File Out (FO) directive.

DETERMINING/SETTING THE ACTIVE LEVEL

The active level is the priority level currently in effect. Directives relating to specific task context are effective only on the active level. You must establish a level as the active level by specifying the Set Level (SL) directive before using any of the memory and register control directives from the directive input device. Thereafter, the active level assumes the value that will most probably be needed, based on the $D DEBUG action in progress; i.e., breakpoint, trace trap, or temporary reference to a different level.

If you want to reference specific task context on another priority level from the directive device, you can change the active level by respecifying the Set Level (SL) directive or temporarily designate another level as the active level by specifying the Set Temporary Level (TL) directive; in the latter case, the level is considered the temporary active level. After the desired actions are performed on the temporary active level, the active level reverts to the level specified in the previous Set Level directive.

The following are guidelines for determining which level is the active level, and methods of setting the active and temporary active level.

1. The Set Level (SL) directive sets (or changes) the active level. The specified level becomes the default level accessible by the operator terminal or another terminal that is the directive input device.

ASSIGN

Assign

Assigns a specified hexadecimal value to a specified symbol. This directive is used to alter registers of the active level and to define reserved symbols. Bound unit breakpoints lie within the Loader, not in your task context. As a result, the Assign directive on a register is refused by $D DEBUG, if the current level's task is stalled on a bound unit breakpoint.

FORMAT:

AS Δsym Δexp {Δsym Δexp...}

ARGUMENTS:

sym

A reserved symbol G through Z or a register.

exp

An expression that resolves to a hexadecimal value of up to 32 bits. The rightmost 20 bits are used for an address register ($Bn) or for the program counter ($P); the rightmost 16 bits are used for all other registers.

Example:

```
AS $R1 -2 X 1408 $B7 X+15
```

This example causes -2 to be assigned to data register 1, 1408 to be assigned to the reserved symbol X, and 141D to be assigned to base register 7.

All Registers

Prints contents of all registers for the active level.

FORMAT:

AR {/lrn}

ARGUMENT:

/lrn

Logical resource number of the device on which the printout will occur.

Default: Current $D DEBUG output device.

Example:

AR/3

This example causes the contents of all the registers for the active level to be printed on the device referred to as logical resource number 3.

NOTE

References to registers on the active level are valid only if the task has come to a true breakpoint. Registers displayed at the time of a bound unit breakpoint are not those of the specified bound unit.

CLEAR ALL BOUND UNIT BREAKPOINTS

Clear All Bound Unit Breakpoints

Clears all bound unit breakpoints.

FORMAT:

CB*

Example:

CB*

This directive clears all bound unit breakpoints of the active level.

Change Memory

Changes the contents of a single specified memory location, or consecutive locations starting at that location, to specified value(s).

FORMAT:

CHΔexpΔrexp{Δrexp...}

ARGUMENTS:

exp

First or only location whose contents will be changed.

rexp

Value(s) to be put in memory location(s).

Example 1:

```
CH 200 4FFF 1716
```

Execution of this directive puts the value 4FFF into location 200 and 1716 into location 201.

Example 2:

```
CH 100 0/10
```

In this example, locations 100 to 10F will be zero-filled.

Example 3:

```
CH 2000 0/10 1/10 2/10
```

This example shows how multiple repeat factors can be used: execution of this directive causes locations 2000 to 200F to be given a value of zero; locations 2010 to 201F to be given a value of 1; and locations 2020 to 202F to be filled with 2s.

Clear Bound Unit Breakpoint

Clears a specified breakpoint for a bound unit.

FORMAT:

CBn

ARGUMENT:

n

Bound unit breakpoint to be cleared; must be a decimal digit from 0 through 9.

Example:

CB3

This directive causes bound unit breakpoint number 3 to be cleared for the bound unit previously defined by SB3.

Clear All True Breakpoints

Clears all defined true breakpoints.

FORMAT:

```
C*
```

Example:

```
C*
```

This directive clears all true breakpoints of the active level.

CONDITIONAL EXECUTION

Conditional Execution

Allows a set of conditions to be tested prior to execution of other $D DEBUG directives. The IF directive is intended to be used in a stored breakpoint directive line. It permits breakpoints to be reported without suspending the active level if the specified condition does not exist. When a breakpoint occurs for which an IF directive has been specified, the following actions occur:

- Any directives preceding IF are executed.
- The IF conditions are evaluated, as follows:

  If TRUE, a line in the following format is displayed on the current $D DEBUG output device

  "exp $\left\{\begin{matrix}\wedge \\ \\ \end{matrix}\right\}\left\{\begin{matrix}< \\ = \\ >\end{matrix}\right\}\left\{\begin{matrix}, \\ \end{matrix}\right\}$ hhhh..."

  and any directives following IF are executed. If a GO directive (described below) does not follow, the active level is suspended.

  If FALSE, no display occurs, and the directives following IF are not executed. The active level continues processing.

FORMAT:

IF exp $\left\{\begin{matrix}\wedge \\ \\ \end{matrix}\right\}\left\{\begin{matrix}< \\ = \\ >\end{matrix}\right\}\left\{\begin{matrix}, \\ \end{matrix}\right\}$ hhhh....;

ARGUMENTS:

exp

Memory address of a byte string argument. This must specify an address; $Rn (where $0 \leq n \leq 7$) cannot be used for exp. (No check for this error is performed, however.)

Clear True Breakpoint

The Clear True Breakpoint (Cn) directive clears a specified true breakpoint.

FORMAT:

Cn

ARGUMENT:

n

Number of the true breakpoint; must be from 0 through 31 (decimal).

Example:

C3

This directive causes true breakpoint number 3 to be cleared.

If both conditions are met, the memory locations 42D1 through 43D0 are dumped to the output device associated with LRN 5, and the active level will continue, in response to the GO directive. If either condition is not satisfied, the dump will not occur, and the active level will continue without suspension.

NOTE

The IF directive can be entered from the terminal, in which case its action corresponds to its entry in a stored directive line. However, using the IF directive from the terminal is of limited usefulness, since the conditions to be tested can be checked by using other directives (e.g., DH).

{^}{<}
{ }{=}
{ }{>}

Specifies the condition to be tested when comparing the memory byte string value to the test parameter. {^} optionally specifies logical negation; i.e., not less than, not equal, not greater than.

{,}

Indicates that the argument is right-byte aligned.

hhhh...

The test parameter, expressed in ASCII as a dense string of pairs of hexadecimal digits; each pair represents one byte. The test parameter may not be an assigned symbol (see the Assign directive (AS)). The length of the parameter is limited by the maximum size of a $D DEBUG stored directive (127 bytes). The parameter's ASCII value must consist of pairs of hexadecimal values. If an odd number of hexadecimal values is specified, a command error is reported when the directive is executed and the task remains suspended to allow for correction.

The IF directive terminator must be a semicolon (;).

Example:

Assume that breakpoint 2, as defined below, is encountered and that $B7 points to memory location 555F:

```
S2 135E (IF 1000^>,3E;IF $B7=42D1;DP/5  $B7/100;GO)
```

In this example two conditions must be true before the Dump (DP) directive (described below) is executed:

1. The rightmost byte at memory location 1000 must be less than or equal to 3E.

2. The byte string found at memory location 555F must be equal to 42D1.

DEFINE TRACE

Define Trace

Associates the directive line within the parentheses with the occurrence of a trace trap or a BRK instruction not already defined as a breakpoint. The specified directive line is stored in the file DEBUG.WORK for future use. The entire define trace directive may comprise a maximum of 126 characters.

When you reuse a disk that has predefined directive lines from a previous execution, the lines may be referred to without redefining them. (See the Set Trace Breakpoint Directive.)

FORMAT:

DTΔ(directive line)

ARGUMENTS:

(directive line)

Directive line comprising maximum of 126 characters.

Example 1:

DT (AR)

This directive causes all registers to be displayed each time a trace trap occurs. (See the All Registers Directive.)

Example 2:

DT ( )

This directive cancels the predefined trace directive line.

Define Directive

Defines a specified directive line for future use and associates that line with a specified number. The directive line is stored on the file DEBUG.WORK and can be referred to by specifying in an Execute (En) directive (described below) the number with which it was associated. The entire Define directive may comprise a maximum of 126 characters.

When you reuse a disk that has predefined directive lines from a previous execution, the lines may be referred to without redefining them. (See the Set True Breakpoint Directive.) This prevents complex predefined directive lines from being respecified each time the system is reloaded for debugging the same problem.

FORMAT:

DnΔ(directive line)

ARGUMENTS:

n

Number with which the specified directive line is associated; must be from 0 through 9.

(directive line)

One or more directives stored for future use.

Example 1:

D3 (CH 100 0)

This example associates the number 3 with the directive within the parentheses. Hereafter, each time the directive E3 (see "Execute Directive") is executed, the parenthetical directive will be executed and location 100 will be zero-filled.

Example 2:

D4 ( )

By storing a null directive line, this example deactivates a previously defined directive line 4 which no longer is required.

DUMP MEMORY

Dump Memory

Displays an area of memory starting at a specified location on the operator terminal or on another specified device. The printout comprises a minimum of eight locations, and is in hexadecimal and ASCII notations.

If the printout is written to a terminal and a value equal to or greater than 121 decimal was specified in the LL directive, 16 locations will be printed on each line.

NOTE

Up to 32K words of memory can be dumped in response to a single DP directive. Dumps of more than 32K must be performed as separate operations.

FORMAT:

DP {/lrn} Δrexp Δrexp...

ARGUMENTS:

/lrn

Logical resource number of the device on which the display occurs.

Default: Current $D DEBUG output device.

rexp

Memory location(s) whose contents are displayed. The display is always in a multiple of eight locations.

Example 1:

DP 200

Execution of this directive displays one line of memory in both hexadecimal and ASCII, starting at location 200.

Display Memory

Displays one or more specified memory locations in hexadecimal notation either on the operator terminal or on another specified device.

FORMAT:

DH {/lrn} Δrexp Δrexp...

ARGUMENTS:

/lrn

Logical resource number of the device on which the information is displayed.

Default: Current $D DEBUG output device.

rexp

Location(s) whose contents are displayed. A minimum of one location may be displayed.

Example 1:

```
DH 200
```

Execution of this directive displays on the current output device the contents of location 200.

Example 2:

```
DH/2 200/100
```

Execution of this directive displays the contents of location 200 to 2FF on the device associated with LRN 2.

End Trace

Disables the j-mode trace for a specific task on the next trap. The trace must first have been enabled using the ST directive.

FORMAT:

```
ET
```

Example:

```
ET
```

This example disables the j-mode trace on the next trap.

Example 2:

```
DP/4 80/3C 200/240
```

This directive causes the contents of locations 80 to BF, and 200 to 43F to be displayed on the device associated with LRN 4. Although location 3C was specified in the directive, the display is through location BF because displays always are in multiples of eight locations.

FILE OUT

File Out

Redirects output from the default $D DEBUG user-in terminal to an alternate device, which must be either a printer or another KSR-compatible terminal. This directive allows messages that result when a breakpoint or other condition occurs to be sent to a device other than the terminal. It has no effect on input to $D DEBUG.

FORMAT:

FO lrn

ARGUMENT:

lrn

Logical resource number associated with the printer or terminal to which output is redirected. The lrn specified overrides any lrn previously specified, and remains in effect until another FO directive is issued or until $D DEBUG is terminated. However, stored directive lines that include /lrn parameters take precedence over the FO directive. That is, the value specified for /lrn in a stored directive line is used instead of the lrn specified in FO.

NOTE

There is no validation of the lrn specified. Thus, if an inappropriate device (e.g., diskette) is specified, no error message is issued to inform the user.

Example:

FO 2

Output is redirected from the terminal to the device associated with lrn 2.

Execute

Retrieves and executes a specified predefined directive line. This directive may not be embedded in Define (Dn) directive lines; it is permitted in Set True Breakpoint (Sn), Define Trace (DT) and Set Bound Unit Breakpoint (SBn) directive lines. (These directives are described elsewhere in this section.)

FORMAT:

En

ARGUMENT:

n

Number of the line to be executed; must be from 0 through 9.

Example 1:

```
D3 (CH 100 0)
E3
```

The directive E3 causes the retrieval and execution of line 3, which was previously defined in the Define directive as CH 100 0.

Example 2:

```
D3 (CH 100 0)
S1 100 (E3)
```

In this example, the Execute directive E3 is embedded in a Set True Breakpoint directive line. The Execute directive will cause the retrieval and execution of line 3, which was previously defined in the Define directive as CH 100 0, whenever true breakpoint 1 is encountered.

LINE LENGTH

Line Length

Maximum line length of each line entered through the operator terminal or another terminal in use.

FORMAT:

LLΔvalue

ARGUMENT:

value

A hexadecimal number between 1E (decimal 30) and 7E (decimal 126).

Example:

LL 48

This directive signifies that the operator terminal or other terminal in use has a maximum line length of 72 decimal characters.

GO

The GO directive resumes execution on the current active level after a breakpoint and can optionally specify a limit-to-pause counter value which applies to j-mode trace traps (see the Start j-mode Trace Directive).

FORMAT:

GO $\left\{ \begin{matrix} \\ \Delta LLLL \end{matrix} \right\}$

ARGUMENT:

LLLL

ASCII expression of 1 to 4 hexadecimal digits yielding a value greater than zero. If used, the ASCII expression is preceded by one space.

Default: 1.

Example:

```
S0 100 (DH 200/10; GO)
```

The task encountering true breakpoint 0 will trap; the associated directive line will be executed by $D DEBUG and the last directive of the directive line (GO) will cause the task to be reactivated.

LIST ALL TRUE BREAKPOINTS

List All True Breakpoints

Lists all currently defined true breakpoints, their locations in memory, and the instruction being replaced. Stored directive lines, if any, are not displayed.

FORMAT:

L* {/lrn}

ARGUMENT:

/lrn

Logical resource number of the device on which printout will occur.

Default: Current $D DEBUG output device.

Example of Listing:

```
BREAKPOINTS
1 LOC = 00ABCD INST = 0F02
```

In this example, true breakpoint 1 is set.

List All Bound Unit Breakpoints

Displays all currently active bound unit breakpoints.

FORMAT:

LB* { /lrn }

ARGUMENT:

/lrn

Logical resource number of the device on which the listing will occur.

Default: Current $D DEBUG output device.

Example of Listing:

```
BU0 LS
BU2 LWD
```

In this example, bound unit breakpoints 0 and 2 have been previously set.

LIST TRUE BREAKPOINT

List True Breakpoint

Displays the directive line associated with a specified true breakpoint.

FORMAT:

Ln {/lrn}

ARGUMENTS:

n

Number of the true breakpoint whose directive line will be listed; must be 0 through 31 (decimal).

/lrn

Logical resource number of the device on which printout will occur.

Default: Current $D DEBUG output device.

Example:

L2/4

This directive causes the display of the directive line of true breakpoint 2 on the device associated with LRN 4.

List Bound Unit Breakpoint

Displays the directive line associated with a specified bound unit breakpoint.

FORMAT:

LBn {/lrn}

ARGUMENTS:

n

Number of the bound unit breakpoint for which the directive line is to be listed; must be 0 through 9.

/lrn

Logical resource number of the device on which the directive line will be listed.

Default: Current $D DEBUG output device.

Example:

LB3/4

This directive lists the directive line associated with bound unit breakpoint 3. The listing occurs on the device associated with logical resource number 4.

PRINT ALL

Print All

Displays all lines predefined by Define (Dn) directives.

FORMAT:

P* {/lrn}

ARGUMENT:

/lrn

Logical resource number of device on which printout will occur.

Default: Current $D DEBUG output device.

Example:

P*/4

This example prints all the directive lines previously defined by Define directive. The printout occurs on the device whose logical resource number is 4.

Print

Displays a specified line predefined by a Define (Dn) directive.

FORMAT:

Pn {/lrn}

ARGUMENTS:

n

Number of predefined line to be printed; must be 0 through 9.

/lrn

Logical resource number of device on which printout will occur.

Default: Current $D DEBUG output device.

Example:

P9/4

This example prints the directive line previously specified by Define directive 9. The printout occurs on the device whose logical resource number is 4.

Print Hexadecimal Value

Prints the value, in hexadecimal, of each specified expression.

FORMAT:

VH {/lrn}Δexp Δexp...

ARGUMENTS:

/lrn

Logical resource number of device on which printout will occur.

Default: Current $D DEBUG output device.

exp

Expression whose value is displayed.

Example:

VH .+100-M

This directive causes the display of the result of the computation defined by the last referenced memory location plus 100 (hexadecimal) minus the value assigned to the temporary symbol M.

Example:

VH X-20

This directive causes to be displayed the result of subtracting 20 (hexadecimal) from the value currently assigned to the temporary symbol X.

Print Header Line

Prints a specified header line starting at the head of form or after a specified number of lines are skipped. The main uses of the print header line directive are to document printed information related to breakpoint or trace trap debugging, and to annotate a line printer memory dump.

FORMAT:

Hn {/lrn} Δ(headerΔ)

ARGUMENTS:

n

Number of lines skipped before header line is printed; can be 0 through 9. 0 causes header to be printed at head of form.

/lrn

Logical resource number of device on which printout will occur.

Default: Current $D DEBUG output device.

(header)

Any ASCII character and/or expressions; each expression must be preceded by a percent (%) sign. If a percent sign is to be printed, two percent signs must be used (%%). A header line must end with a space character; i.e., there must be a space immediately before the right parenthesis. Left and right parentheses must be balanced within header lines.

Example:

H0/2 (DUMP OF BREAKPOINT FOR LEVEL %$S)

This example illustrates a way to document dumps. As soon as a carriage return is typed, the above header will be printed at the top of a new page on the device identified by logical resource number 2.

QUIT

Quit

Clears all breakpoints, closes the work file DEBUG.WORK, and disables the $D DEBUG trap handler before effectively aborting the $D DEBUG task group. If the group is aborted by a directive other than QT, the results are unspecified.

FORMAT:

```
QT
```

Example:

```
QT
```

This directive clears all breakpoints, closes the work file DEBUG.WORK, disables the $D DEBUG trap handler, and then aborts the $D DEBUG task group.

Print Trace

Displays a pre-defined trace directive line.

FORMAT:

PT {/lrn}

ARGUMENT:

/lrn

Logical resource number of device on which printout will occur.

Default: Current $D DEBUG output device.

Example:

PT/4

This example prints the directive line previously defined in a Define Trace directive. The printout occurs on the device whose logical resource number is 4.

SET BOUND UNIT BREAKPOINT

Set Bound Unit Breakpoint

Sets a numbered breakpoint for a specified bound unit or bound unit overlay. A given bound unit (BU) breakpoint refers to either roots or to overlays, or to both. When a bound unit breakpoint is encountered, a message informs the user where the bound unit or overlay has been loaded into memory. True breakpoints can then be set at specified locations in the program. Because a bound unit is loaded at the time the task associated with it is created, the level number displayed when a BU breakpoint occurs is not necessarily the one used when requests for that task are later executed.

The entire Set Bound Unit Breakpoint directive may comprise a maximum of 127 characters.

The message format is:

```
*BU n $SL=00xx $E=00xxxx + 00xx
```

n

Number of bound unit breakpoint; must be 0 through 9.

$SL=00xx

Priority level.

$E=00xxxx + 00xx

Bound unit base address plus entry point offset as defined by the bound unit or by the caller. Used in place of $P associated with true breakpoints.

FORMAT:

```
                 ( bound-unit-name                  )
                 ( bound-unit-name/overlay-number   )
SBnΔ             ( bound-unit-name/*                ) Δ(directive line)
                 ( */overlay number                 )
                 ( */*                              )
```

ARGUMENTS:

n

Bound unit breakpoint number; must be from 0 to 9.

Reset File

Prohibits execution of directives that use the file DEBUG.WORK until another specify file (SF) directive is issued. The directives that use DEBUG.WORK are: P*, Pn, PT, Sn, En, Dn, DT and SBn.

FORMAT:

```
RF
```

Example:

```
RF
```

This directive prohibits execution of directives using the file DEBUG.WORK. You must issue another specify file (SF) directive before you use those directives (listed above).

SET LEVEL

Set Level

Sets the active priority level to a specified value. This level remains in effect until another SL directive is issued. The level may be temporarily changed via the Set Temporary Level (TL) directive (described below).

FORMAT:

SL␣exp

ARGUMENT:

exp

Number of active priority level in hexadecimal notation.

Default: 0

Example 1:

```
SL C
```

This directive sets the active priority level to 12 (decimal). If the All Registers directive is entered after the SL directive, the registers on level 12 are displayed.

Example 2:

This example shows how the active level can be designated, permanently changed, and temporarily changed.

```
SL C      The active level is 12 (decimal)
 .
 .
 .
SL A      The active level is 10 (decimal)
 .
 .
 .
TL B;AR   The active level is temporarily set to 11
          (decimal). After the AR directive is executed,
          the active level reverts to level 10 (the level
          specified in the last SL directive).
```

bound-unit-name

Name of the bound unit to which the breakpoint applies; up to six ASCII characters (first six characters of the bound unit name).

overlay-number

Hexadecimal number of the bound unit overlay.

*

"All" roots or "all" overlays, depending on context.

(directive line)

Directives to be executed when the bound unit/overlay is loaded.

Example:

```
SB6 SOOZ/A (IF 3D02=5354;VH X-2;GO)
```

This directive sets bound unit breakpoint 6 for overlay number A of the bound unit named SOOZ. The directive line specifies that if the condition indicated is true (the byte string at location 3D02 equals 5354), then the value of the temporary symbol X minus 2 is displayed. When overlay A is loaded into memory, its location is displayed at the terminal, and the directive line associated with bound unit breakpoint 6 is executed.

SET TRUE BREAKPOINT

Set True Breakpoint

Sets a numbered true breakpoint at a specified location. When the true breakpoint is encountered, there is a typeout indicating the contents of the location counter and the active priority level; task execution is suspended; and the stored directive line, if there is one, is executed. The Set File (SF) directive is a precondition for directive line execution. The entire Set True Breakpoint directive may comprise a maximum of 126 characters.

If there is a preexisting directive line associated with a given true breakpoint and that directive line is no longer applicable, clear the line by designating empty parentheses ( ) when setting the breakpoint.

The message format is:

($D) BPn SP=00xxxx $SL=00xx

$P=00xxxx

Location counter

$SL=00xx

Priority level

NOTES

1. If a true breakpoint is set in any of the following types of instructions, that breakpoint must be cleared (using the Cn directive) before continuing execution (GO) directive): input/output, generic (BRK), scientific, LEV, invalid instruction, or instruction with an invalid address syllable. You may avoid this restriction by clearing the existing true breakpoint with a stored directive and then resetting it in the stored directive line of a subsequent Set True Breakpoint directive, as shown in Example 3.

2. A GO directive embedded in a Set True Breakpoint directive line allows task execution to proceed after the desired operations have been performed, without further operator intervention.

Set Temporary Level

Sets the active priority level to a temporary specified value. The level specified in the TL directive remains in effect until an SL or another TL directive is issued, or until the end of the directive line. If the end of the line is reached before another SL or TL directive is encountered, the value specified in the last SL directive becomes the active priority level.

FORMAT:

TLΔexp

ARGUMENT:

exp

Value designating the temporarily active priority level.

Example:

```
SL 20
TL A;AR
TL B;AR
```

The first TL directive designates level 10 as the temporarily active priority level so that all registers on that level can be displayed via the subsequent AR directive.

The second TL directive designates level 11 as the temporarily active priority level so that all registers on the level can be displayed via the subsequent AR directive.

After the last TL directive is executed, the active level will be 32 (decimal): the level specified in the last SL directive.

Specify File

Identifies the device on which the file DEBUG.WORK is located. Since the function of the SF directive is to find the work file, it should be the first directive executed; failure to do this results in the issuing of an error message as soon as a directive that requires the work file is used.

FORMAT:

SFΔlrn

ARGUMENT:

lrn

Logical resource number of the disk device on which the file DEBUG.WORK is located; must be specified in hexadecimal notation. This lrn must be the same as that of the bootstrap device.

Example:

SF 1

This example specifies that the work file is on the device whose logical resource number is 1.

FORMAT:

$$Sn\Delta exp\left\{\begin{matrix}\\ \Delta(directive\ line)\\ \\ \end{matrix}\right\}$$

ARGUMENTS:

n

Number of true breakpoint; must be 0 through 31 (decimal).

exp

Location at which true breakpoint will occur.

(directive line)

Directive(s) that will be executed when the true breakpoint is encountered.

Example 1:

```
S0 100 (DH 200/10;GO)
```

This directive will cause the display of locations 200 to 20F when location 100 is reached and the task will proceed.

Example 2:

```
S0 100 ( )
```

This directive cancels any directive line previously associated with true breakpoint 0 and sets true breakpoint 0 at line 100.

Example 3:

```
S0 1000 (AR;C0;GO)
S1 1003 (S0 1000;GO)
```

The first directive line sets true breakpoint number 0 at location 1000, causes a printout of all registers on the active level, and then clears true breakpoint number 0 because the instruction at location 1000 is restricted (see Note 1 above).

The second directive line sets true breakpoint number 1 at location 1003 and then reestablishes true breakpoint 0 at location 1000; the second true breakpoint line causes no visible action except the printing of the breakpoint message.

SAMPLE SD DEBUG SESSION

A sample DEBUG session is shown below to illustrate some of the directives and procedures described earlier in this section.

The bound unit being debugged is TSTNOW, listed in Figure 17-1. The debugging session is shown in Figure 17-2.

Start j-mode Trace

Sets the given task's M1 register j-bit on. As a result, any departure from the current processing sequence will cause a trap. $D DEBUG treates the trap as a "trace trap." The following points apply:

- j-mode trace can be started only for a task which is currently suspended due to a true breakpoint.
- The Start j-mode Trace directive will be refused if the task is suspended due to a bound unit breakpoint.
- j-mode processing is specific to a given task and is shut off or restored at the monitor call interfaces.
- When a task is running in j-mode, $D DEBUG's handling of successive traps is governed by the limit-to-pause counter of the GO directive.
- Limit-to-pause has a default value of 1, but may be set to an arbitrary value by the GO directive. $D DEBUG decrements the limit-to-pause once for each occurrence of a trace trap. When limit-to-pause assumes the value zero, the trapped task is suspended to permit operator action. When the task is reactivated (GOΔ[LLLL]) the limit-to-pause is reset to the default value or to a user-specified value.

FORMAT:

```
ST lvl
```

ARGUMENT:

lvl

The active level.

Example:

```
ST C
```

This example sets the task's M1 register j-bit on at active level 12.

```
                TITLE TSTNOW          08/13/82  1334.4 edt Fri       HRS ASSEMBLER 9.00  -LAF  PAGE  0002
000046                                          *           $USOUT    !SLEW_3,=20,=1
000047                                          *           LDB       $b4,!SLEW_3
000048   0028   CBC0 0036                                   lab       $b4,slew_3
000049   002A   E870 0014                                   ldr       $r6,=20
000050   002C   F870 0001                                   ldr       $r7,=1
000051   002E   0001                                        mcl
000052   002F   0801                                        dc        x'0801'
000053   0030   1903                                        bez       $r1,>bb
000054                                          *
000055   0031   A851                            aa          ldr       $r2,=$r1
000056   0032   0F82                                        b         >cc
000057   0033   2C00                            bb          ldv       $R2,0
000058                                          *
000059                                          *$c         $TRMRQ
000060          0034                            cc          equ       $
000061   0034   0001                                        mcl
000062   0035   0103                                        dc        x'0103'
000063                                          *
000064   0036   2041                            slew        dc        z'2041'
000065   0037   584E 4F57 5020                  buffer      text      a'[NOW] '
000066   003A   2020                                        resv      11,z'2020'
000067   0045   2020                            extra       resv      5,z'2020'
000068                                          *
000069   004A   2041                            slew_2      dc        z'2041'
000070   004B   5855 5345 525D                  buff_2      text      a'[USER]'
000071   004E   2020                                        resv      12,z'2020'
000072   005A   2020                            extr_2      resv      5,z'2020'
000073   005F   2041                            slew_3      dc        z'2041'
000074   0060   586E 6F74 205B                  buff_3      text      a'[not [equal 2 2]] '
         0063   6571 7561 6C20
                3220 325D 5020
000075   0069   2020                                        resv      12,z'2020'
000076                                          *
000077   0075   0000                                        end       tstnow,begin
0000   ERR COUNT
```

Figure 17-1 (cont). Listing of TSTNOW

```
     TITLE TSTNOW          08/13/82  1334.4 edt Fri       HRS ASSEMBLER 9.00  -LAF  PAGE  0001
000001                                          title tstnow
000002                                 *
000003                                 *        COPYRIGHT, (C), 1981, HONEYWELL INFORMATION SYSTEMS,
000004                                 *
000005  0000  CBC0 0036                begin    lab      $b4,buffer
000006  0002  E870 401C                         ldr      $r6,=Z'401C'
000007  0004  1C05                             ldv      $r1,5
000008                                 *
000009  0005  0001                             mcl
000010  0006  0C08                             dc       x'0C08'
000011                                 *
C00012  0007  19AA                             bnez     $r1,>aa
000013                                 *
C00014                                 *        $USOUT   !SLEW,=29,=1
000015                                 *        $LDB     $B4,!SLEW
C00016  0008  CBC0 002D                         lab      $b4,slew
000017  000A  E870 001D                         ldr      $r6,=29
C00018  000C  F870 0001                         ldr      $r7,=1
000019  000E  0001                             mcl
C00020  000F  0801                             dc       x'0801'
000021                                 *
000022        0010                     user     equ      $
000023  0010  CBC0 003A                         lab      $b4,buff_2
C00024  0012  E870 401E                         ldr      $r6,=Z'401E'
000025  0014  1C05                             ldv      $r1,5
000026  0015  0001                             mcl
000027  0016  0C08                             dc       x'0C08'
000028  0017  199A                             bnez     $r1,>aa
C00029                                 *        $USOUT   !SLEW_2,=31,=1
000030                                 *        LDB      $B4,!SLEW_2
000031  0018  CBC0 0031                         lab      $b4,slew_2
000032  001A  E870 001F                         ldr      $r6,=31
000033  001C  F870 0001                         ldr      $r7,=1
000034  001E  0001                             mcl
000035  001F  0801                             dc       x'0801'
000036                                 *
C00037        0020                     nest     equ      $
C00038  0020  CBC0 003F                         lab      $b4,buff_3
C00039  0022  E870 4011                         ldr      $r6,=Z'4011'
C00040  0024  1C05                             ldv      $r1,5
C00041  0025  0001                             mcl
C00042  0026  0C08                             dc       x'0C08'
000043                                 *
000044  0027  198A                             bnez     $r1,>aa
000045                                 *
```

Figure 17-1. Listing of TSTNOW

```
17  GO
    ($D)*BP  2  $SL=001E $P=04036A
    ($D)04036A/ 0002 CBC0 002D E870 001D F870 0001 0001  .....-.P...P....
    ($D)040397/ 2041 4672 6920 4175 6720 3133 2C20 3139   AFRI AUG 13, 19
    ($D)0403A7/ 3832 2020 2020 3135 3A33 303A 3030 2020  82     15:30:00
    ($D)$R1=0000  $R2=001C  $R3=0000  $R4=0000  $R5=0000  $R6=001C
    ($D)$R7=0000  $B1=000000  $B2=040404  $B3=000000  $B4=04039A
    ($D)$R5=002F7C  $B6=040363  $B7=040469  $P=04036A  $I=0000  $S=401E
18  SS X+17 (AR:DP X+4A/10)
19  GO
    ($H)FRI AUG 13, 1982     15:30:00
    ($D)*BP  3  $SL=001E $P=04037A
    ($D)$R1=0000  $R2=0008  $R3=0000  $R4=0000  $R5=0000  $R6=0008
    ($D)$R7=0000  $B1=000000  $B2=040284  $B3=000000  $B4=0403AE
    ($D)$R5=002F7C  $B6=040363  $B7=040469  $P=04037A  $I=0000  $S=401E
     $D)0403AD/ 2041 4849 532E 4C36 2E41 2020 2020 2020   AHIS.L6.A
    ($D)040365/ 2020 2020 2020 2020 2020 2020 2020 2020
20  SS X+27
21  L*
    ($D)BREAKPOINTS
    ($D)  1  LOC=040363 INST=CBC0
    ($D)  2  LOC=04036A INST=19AA
    ($D)  3  LOC=04037A INST=199A
    ($D)  4  LOC=04038A INST=198A
22  L4
    ($D)CMD INACTIVE "L4"
23  GO
    ($H)HIS.L6.A
    ($D)*BP  4  $SL=001E $P=04038A
24  AR
    ($D)$R1=0805  $R2=0009  $R3=0000  $R4=0000  $R5=0000  $R6=0000
    ($D)$R7=0000  $B1=000000  $B2=040284  $B3=000000  $B4=0403C3
    ($D)$R5=002F7C  $B6=040363  $B7=040469  $P=04038A  $I=0000  $S=401E
25  DP X+5F
    ($D)0403C2/ 2041 586E 6F74 205B 6571 7561 6C20 3220   A[NOT [EQUAL 2
26  DP X+5F/10
    ($D)0403C2/ 2041 586E 6F74 2058 6571 7561 6C20 3220   A[NOT [EQUAL 2
    ($D)0403CA/ 325D 5020 2020 2020 2020 2020 2020 2020  2]
27  GO
    ($H)EC: 17 0805 (1D)
    ($H)UNBALANCED QUOTATION MARKS, BRACKETS, OR PARENTHESES EXIST.  THE NEXT FUNCTION
    ($H)IS PERFORMED.  CORRECT THE DELIMITER AND RETRY.
    ($H)RDY:
```

Figure 17-2 (cont). Sample Debugging Session

```
($S)GC056 M0D400-L3.0-07/02/0741
($S)  "CLMIN...OR *"
($S) C>
GUI!
($S)             ** M4/3.0    V10.1 **
($S)THIS SYSTEM IS BUILT ON THE NEW DIRECTORY STRUCTURE
($S)DATE TIME (E.G., 81 JAN 10 1405):  82 AUG 13 1525
($S)FRI AUG 13, 1982    15:25:08
($P)DAEMON GROUP READY!
($H)GROUP READY'
($H)$H
1  EC GROUP$D
   ($D) DEBUG-R300-05/25/0930
2   C :$D:
3  SF 1
4  SB1 TSTNOW ($1 @@@DP $E)
5  LB1
   ($D)BU1  (DP $E)
6   $H ATERRY>LAF>TSTNOW
   ($D)*BU 1 $SL=001E $E=040363 +  0000
   ($D 0+0363/ C8C0 0036 E870 401C 1C05 0001 0C08 19AA  ...6.P@.........
7  SL 1E
8  L*
   ($D)INACTIVE BP "L*"
9  B1 $E (AR)
10 L!
    $E BREAKPOINTS
   ($D) 1  LOC=0+0363 INST=C8C0
11 L!
   ($D) B1  (AR)
12 GO
   ($D)*BP  1  $SL=001E $P=040363
   ($D)$R1=0000  $R2=0000  $R3=0000  $R4=0000  $R5=0000  $R6=0000
   ($D)$R7=0075  $B1=000000  $B2=040469  $B3=000000  $B4=040463
   ($D)$B5=002F7C  $B6=040363  $B7=040469  $P=040363  $I=0000  $S=401E
13 AR X $P
14 DP X+7
   ($D)040366/ 19AA C8C0 002D E870 0010 F870 0001 0001  .....-.P...P....
15 >2 X+7 (DP $P;DP X+36/10;AR)
16 L*
   ($D)BREAKPOINTS
   ($D) 1  LOC=040363 INST=C8C0
   ($D) 2  LOC=0+036A INST=19AA
```

Figure 17-2. Sample Debugging Session

Each numbered directive in Figure 17-2 is explained below by a correspondingly-numbered comment.

1. Invoke the system-supplied EC file (GROUP$D.EC) to load $D DEBUG.

2. Change the default group id to $D so that you will not have to precede every line of input with '$D'.

3. Open the DEBUG work file DEBUG.WORK.

   The work file must reside on the boot volume; the logical resource number of the boot device, specified by the SF directive, is 1.

4. Set a bound unit breakpoint on the program TSTNOW, specifying a directive line to be stored in the work file. This directive line will be executed each time the bound unit breakpoint is encountered.

5. List bound unit breakpoint 2.

6. From the $H group, invoke the program TSTNOW. This invocation causes bound unit breakpoint 1 to be encountered and the breakpoint message to be displayed. Execution of the stored directive line (specified in step 4) dumps a line of memory, starting at the location associated with the symbol $E.

7. Set the level to that shown in the breakpoint message.

8. List all true breakpoints currently set. As the returned message indicates, no true breakpoints have yet been set.

9. Set true breakpoint 1 at the location associated with $E and specify a directive line that will be stored in the work file for future execution.

10. List all true breakpoints currently set.

11. List the stored directive line associated with true breakpoint 2.

12. GO from the bound unit breakpoint 1. This directive initiates execution of TSTNOW. True breakpoint 1 is encountered, its breakpoint message is displayed, and its associated directive line is executed.

13. Assign the current value of $P (program counter) to the temporary symbol X.

```
28  L*
    ($D)BREAKPOINTS
    ($D) 1  LOC=040363 INST=CBC0
    ($D) 2  LOC=04036A INST=19AA
    ($D) 3  LOC=04037A INST=199A
    ($D) 4  LOC=04038A INST=198A
29   $H ^TERRY>LAF>TSTNOW
    ($D)*BU 1 $SL=001E $E=040363 +  0000
    ($D)040363/ CBC0 0036 E870 401C 1C05 0001 0C08 19AA  ...6.P8.........
30  VH (
    ($D)X=040363
31  GO
    ($H)FRI AUG 13, 1982    15:35:02
    ($H)HIS.L6.A
    ($H)EC: 17 0805 (1D)
     $H)UNBALANCED QUOTATION MARKS, BRACKETS, OR PARENTHESES EXIST.  THE NEXT FUNCTION
    ($H)IS PERFORMED.  CORRECT THE DELIMITER AND RETRY.
    ($H)RDY:
```

Figure 17-2 (cont). Sample Debugging Session

29. Reinvoke the bound unit TSTNOW from the $H group. Bound unit breakpoint 1 is encountered again, as in step 6.

30. Verify the value of temporary symbol X.

31. GO from bound unit breakpoint 1.

    Note that even though $D DEBUG has listed true break points (step 28), none are set and none are encountered. Unlike bound unit breakpoints, true breakpoints must be reset between multiple invocations of the bound unit being debugged. When the bound unit is reinvoked, a new copy of the bound unit is loaded into memory, overwriting the version containing true breakpoint instructions.

The value of $P is currently the base address of the bound unit. Having assigned to X the value of $P, you can now refer to any location in the bound unit by the expression 'X + offset'. Offsets are shown in column 2 of the bound unit listing (Figure 17-1).

14. Dump one line of memory, starting at offset 7.

15. Set true breakpoint 2 at offset 7, specifying a directive line to be stored in the workfile for future execution.

16. List all true breakpoints currently set.

17. GO from true breakpoint 1. Execution of the bound unit continues. When breakpoint 2 is encountered, the message is displayed and the associated directive line is executed.

18. Set true breakpoint 3 at offset 17, specifying a stored directive line.

19. GO from true breakpoint 2, causing breakpoint 3 to be encountered, the message to be displayed, and the associated directive line to be executed.

20. Set true breakpoint 4 <u>without</u> specifying a directive line. The directive line executed when breakpoint 4 is encountered will be one previously specified and currently stored in the DEBUG workfile.

21. List all the true breakpoints currently set.

22. List the stored directive line associated with breakpoint 4. The message returned indicates that this directive line is null. Therefore, encountering breakpoint 4 will cause only the message to be displayed. (Null directives are explained earlier in this section under the description of the Define Directive (Dn) directive.)

23. Go from breakpoint 3, causing breakpoint 4 to be encountered.

24. Display all registers.

25. Dump one line of memory, starting at offset 5F.

26. Dump two lines of memory, starting at offset 5F.

27. GO from true breakpoint 4. TSTNOW completes execution; the default group reverts to $H, which awaits input (RDY).

28. List all true breakpoints currently set.

- Set or clear bound unit breakpoints to gain control of bound units as they are loaded
- Set or clear quick breakpoints (from the $S group only) to monitor time-dependent tasks without undue distortion of time
- Display, change, and dump either memory or registers
- Evaluate expressions.

INVOKING THE MULTI-USER DEBUGGER

The command used to invoke the Multi-User Debugger is:

```
DEBUG
```

There are no valid arguments with this command.

MULTI-USER DEBUGGER FILE REQUIREMENTS

For true and bound unit breakpoints, Debugger directives can be stored in a user-defined work file. If used, this file must be opened by the Specify File directive (the SF directive is described later in this section) and must always be followed by the suffix ".DB". This work file requires a size of 64 sectors on any media.

The Multi-User Debugger directives associated with quick breakpoints are stored in memory, and optionally, in a work file as described above. Output generated by these directives is written to memory and, optionally, to a user-defined quick disk file. This quick disk file must have been created previously outside the Debugger task using the Create File command (see the Commands manual for details) and must be referenced within the Multi-User Debugger task by a Specify File (SF) directive. This file must end with the suffix ".QK".

The Debugger directives mentioned above are identified and described in Table 18-2, later in this section.

MULTI-USER DEBUGGER MEMORY REQUIREMENTS

To set true or bound unit breakpoints, the reentrant portion of the Multi-User Debugger requires a minimum memory area of 2250_{10} words. The separate data portion of the amount of memory required per group is approximately 1575_{10} words. This includes all non-reentrant code, the Multi-User Debugger overlay area, and all necessary data information.

To debug time-dependent tasks using quick breakpoints, the total amount of memory required is 7500_{10} words (for reentrant and data portions) plus the amount of memory you requested for the quick memory buffers. The quick memory buffers are described later in this section.

MULTI-USER DEBUGGER OPERATION

The Multi-User Debugger is restricted to the write privileges of the group it serves; several users can debug within their own groups without affecting other groups. Since the Multi-User Debugger runs under any user-defined group, memory protection is dependent upon the task group. If no memory protection is established, you can alter any and all memory; therefore, the task should run in a protected environment.

The Multi-User Debugger handles traps to Trap Vector 14 (unauthorized reference to protected memory) and Trap Vector 15 (reference to unavailable resource) and continues as described below.

An error message is displayed if you try to access non-virtual memory within any Multi-User Debugger directive except the Dump Memory (DP) directive. If a Trap-to-Trap-Vector-15 occurs when a DP directive is specified, the Multi-User Debugger dumps as much of the requested memory as possible. Once a nonvirtual address is invoked, the rest of the current line to be printed is blank-filled. The current nonvirtual address is advanced to the value that is the next multiple of 1K. The procedure continues until the area to be dumped is exhausted or the end of memory is reached.

ENTERING DIRECTIVES

Multi-User Debugger directives consist of a directive name only or a directive name and one or more arguments. Within a directive, arguments are separated from each other by one or more spaces. Multiple Debugger directives can be entered on a single line; each directive, except the last, must be followed by a semicolon (;). At the end of each line (i.e., immediately after the last or only directive), press carriage return. Except where otherwise specified, all argument values are entered in hexadecimal notation.

Debugger directives may only be entered when the Debugger has control of the group. This occurs when:

- The Debugger is loaded.
- A breakpoint occurs.
- You press the BREAK key and "DEBUG" is typed as the post-break input. (Break key functionality is described later in this section.)

Special symbols are used in the Multi-User Debugger directive lines. These symbols are described in Table 18-2.

NOTE

The Multi-User Debugger will only recognize tasks which are in a trapped state.

Table 18-1 summarizes Multi-User Debugger directives by function. These directives are described in detail on the following pages.

NOTE

Pay careful attention to the format of each directive, because the use of delimiters, if any, between a directive name and the first (or only) argument varies according to which directive is being specified.

Table 18-1. Summary of Multi-User Debugger Directives by Function

Function	Directive	Directive Name
Directive line and handling	Dn	Define directive line n
	En	Execute directive line n
	P*	Print all predefined directive lines
	Pn	Print predefined directive line n
True breakpoint	C*	Clear all true breakpoints
	Cn	Clear true breakpoint n
	L*	List all true breakpoints and associated directive lines
	Ln	List true breakpoint n and associated directive line
	Sn	Set true breakpoint n
Bound unit breakpoint control	CB*	Clear all bound unit breakpoints
	CBn	Clear bound unit breakpoint n
	LB*	List all bound unit breakpoints and associated directive line
	LBn	List bound unit breakpoint n and associated directive line
	SBn	Set bound unit breakpoint n

Table 18-1 (cont). Summary of Multi-User Debugger Directives by Function

Function	Directive	Directive Name
Quick breakpoint control	CQn	Clear quick breakpoint n
	CQ*	Clear all quick breakpoints
	LQn	List quick breakpoint and its associated directive line
	LQ*	List all quick breakpoints and their associated directive lines
	MQ	Get memory block for quick breakpoint information storage
	PQ	Print pointer to quick memory block
	RQ	Return quick memory
	SQn	Set quick breakpoint n
Trace trap control	DT	Define trace directive line
	PT	Print trace directive line
	ST	Start j-mode trace
	ET	End j-mode trace
Active level control	SL	Set active level
	TL	Set temporary active level
Memory and register control	AR	Print contents of all registers of the active level
	CH	Change memory
	DH	Display memory in hexadecimal
	DP	Dump memory in hexadecimal and ASCII
Symbol control	AS	Assign a hexadecimal value to symbol
	VH	Print value of expression in hexadecimal
General execution	E	Temporary escape to the command processor
	FO	Redirect output
	GO	Continue execution from breakpoint
	Hn	Print header line
	IF	Conditional execution
	MODE	Change from numeric to symbolic mode or vice-versa
	RF	Reset file location
	SF	Specify file location
	SP	Temporarily suspend the Multi-User Debugger; return control to the command processor (sleep)
	QT	Abort Multi-User Debugger task (quit)

Table 18-1 (cont). Summary of Multi-User Debugger Directives by Function

Abnormal Trap Control	CT TB TT	Clear abnormal trap bit Turn on abnormal trap bit Terminate trapped task

NOTE

The memory and register control directives (AR, CH, DH, and DP) apply to registers on the active level. To determine which level is the active level and/or to set the active level to a specified value, see "Determining/Setting the Active Level" below.

Table 18-2. Symbols Used in Multi-User Debugger Directive Lines

Symbol Type	Meaning
Arithmetic Operators	
plus sign (+)	Performs additon.
minus sign (-)	Performs subtraction.
K	Multiplies a hexadecimal integer by 1024 decimal (400 in hexadecimal) when K is the last character of an integer expression.
Address Operators	
period (.)	Represents the last start address used in a previous memory reference directive (DH, CH, DP).
ampersand (&)	Represents the address of the next location beyond the last one used by a previous memory reference directive (DH, CH, DP).
brackets []	Signifies the contents of the location defined by the expression within the brackets. Three levels of nesting may be used.

Table 18-2 (cont). Symbols Used in Multi-User Debugger Directive Lines

Symbol Type	Meaning
Reserved Symbols	
$Bn	Contents of base register n of the active level. The values 1 through 7 can be used for n.
$Rn	Contents of the data register n of the active level. The values 1 through 7 can be used for n.
$P	Contents of the program counter of the active level.
$I	Contents of the indicator register of the active level.
$IV	Address of the Task Control Block of a trapped task which is currently at the head of the active level's trap queue.
$IV_Bn	Represents the contents of the base register n as stored in the Interrupt Save Area (ISA) of the active level. The values 1 through 7 can be used for n.
$IV_Rn	Represents the contents of the data register n as stored in the ISA of the active level. The values 1 through 7 can be used for n.
$IV_Mn	Represents the contents of the Commercial or Scientific Instruction Processor mode control register n as stored in the ISA of the active level. For the Commercial Instruction Processor mode, n must equal 3. For the Scientific Instruction Processor mode n can be either 4 or 5.
$IV_Sn	Represents the contents of the scientific accumulator register n as stored in the ISA of the active level. The values 1 through 3 can be used for n.
$IV_Kn	Represents the contents of the K register n as stored in the interrupt save area (ISA) of the active level. The value of n can be 1 through 7.

Table 18-2 (cont). Symbols Used in Multi-User Debugger Directive Lines

Symbol Type	Meaning
$Kn	Contents of the K register n of the active level. The value of n can be 1 through 7.
$S	Contents of the system status register (level number and privilege bit only) of the active level.
$SL	Represents the value of the level number of the active level.
$E	Represents the entry point of a bound unit as defined in the bound unit or by the caller. This reserved symbol is used only at the time of a bound unit breakpoint, in place of $P associated with true and quick breakpoints.
$T	Represents the address of the stack of the active level.
G through Z	Twenty single-character symbols having initial values of zero. Values may be assigned using the AS directive.
$CI	Represents the contents of the Commercial Processor indicator word of the active level.
$Cl	Represents the contents of the Commercial Processor remote descriptor table of the active level.
$SI	Represents the contents of the Scientific Instruction Processor (SIP) indicator word of the active level.
$Mn	Represents the contents of the mode control register of the active level. The values 1 through 7 can be used for n.
Debug Language:	
{^} < {^} = {^} >	The condition to be satisfied in an IF directive for continuous processing of the directive line. {^} indicates a logical 'NOT' which may optionally be used.
parentheses ( )	Indicate directive or header information to be stored for later use. Unmatched right parentheses result in an error. A right parenthesis that is paired with the first left parenthesis terminates the directive definition.

Table 18-2 (cont). Symbols Used in Multi-User Debugger Directive Lines

Symbol Type	Meaning
Debug Language (cont):	
exp	Indicates a valid expression formed by using expression elements. Expression elements are addresses, reserved symbols, and hexadecimal values up to 32 bits in length. No more than one address is allowed within an expression. An expression element may be preceded by the positive (+) or negative (-) unary operator. Expression elements can be joined by the addition (+) or subtraction (-) operator.
rexp	Consists of exp_1/exp_2, where exp_1 is a hexadecimal number that is a value of a location expression; exp_2 is an optional hexadecimal repeat factor whose value must be between 1 and 32,767. If exp_2 is omitted, there is no repetition.
;	Separation character between directives on the same line.
*	Signifies "all" in certain print, clear, and list directives.

MULTI-USER DEBUGGER AND BREAK KEY FUNCTIONALITY

Typing "DEBUG" as a response to the break key transfers you to the Multi-User Debugger task. To return to the previous stack level, enter the Sleep (SP) directive or terminate the Debugger completely with the Quit (QT) directive. The description of the Debugger and break key functionality applies only to true and bound unit breakpoints, and not to quick breakpoints. Break key functionality is not supported in the $S task group.

If DEBUG was the task that was broken, any command is a valid response, including PI, UW, SR, or NEW_PROC.

NOTE

The Program Interrupt (PI) response will return the user group to the Debugger input level and allow the entry of Debugger directives.

If the Debugger task was broken and "DEBUG" is entered as the response, you are placed in the Debugger input mode.

The Unwind (UW) response will cause the Debugger to execute either the GO or SP directive, depending on which is appropriate at the time of the **BREAK**. If the Debugger was activated as the result of encountering a breakpoint, entering UW causes execution of the GO directive.

PLANNING CONSIDERATIONS

Setting True Breakpoints and Bound Unit Breakpoints

True breakpoints and bound unit breakpoints can be set to trap at selected task code locations. At true breakpoints, memory and register values can be displayed and changed. At bound unit breakpoints, only memory can be displayed and changed. The registers displayed at the time of a bound unit breakpoint are not those of the trapped task. In this way, a task can be executed, the value of its variables checked as execution proceeds, code modified, and if necesary, variable values changed in order to test the sequence of code up to the next breakpoint.

Setting Quick Breakpoints

Quick breakpoints can be set to trap at selected locations to monitor time-dependent functions (for example, monitoring a driver). At these breakpoints, memory and registers can be stored in a block of memory (reserved by means of the Get Quick Memory directive) and, optionally, in a disk file to be retrieved and studied at some later time at your convenience. These breakpoints must be set when you are running in the system task group ($S).

Preliminary Steps for Using Quick Breakpoints

Before invoking the Debugger from the $S task group:

1. Calculate the approximate amount of memory necessary for the quick memory buffers.

2. Create a Debugger quick disk file with the format

   path.QK

   using the Create File (CR) command (see the Commands manual). The quick disk file must be created from a user-defined group. It should be created as a a relative file with a control interval (CI) size greater than or equal to the size of the quick memory blocks that you specify in the Get Quick Memory (MQ) directive.

3. Enter the command

   EC !CONSOLE

   to load the command processor.

Now you can invoke the Multi-User Debugger and monitor the time-dependent task without causing any time distortion within the task.

Guidelines for Setting Breakpoints

- True breakpoints can be set in a bound unit in a task group (or in an overlay of a bound unit in a task group) only when the task group/overlay currently is memory resident. Use the SBn (Set Bound Unit Breakpoint) directive to gain control of a task group bound unit/overlay when it is loaded, to allow true breakpoints to be properly set.

- True breakpoints may not be set in code that will be executed at the inhibit level (level 3).

- True breakpoints are set in task groups by specifying the Set Breakpoint (Sn) directive. (The detailed description of the Sn directive later in this section includes additional rules for specifying true breakpoints.)

- Quick breakpoints can only be set from the system task group ($S); i.e., you must be debugging from the terminal designated as the operator terminal.

- Quick breakpoints are set in the $S task group by specifying the Set Quick Breakpoint (SQn) directive. (The detailed description of the SQn directive later in this section includes additional rules for specifying quick breakpoints.)

- Only quick breakpoints may be set in sharable code.

- Quick breakpoints may be embedded in true or bound unit directive lines. Note that in this case you set all breakpoints from the system task group and that these breakpoints could impact all users. Thus, caution must be taken when debugging in the system task group.

Controlling Output

Output can be redirected by using a true or bound unit breakpoint. When the breakpoint condition occcurs, the FO directive can be used to redirect the output.

When quick breakpoints are utilized, output sent to the previously specified user-defined disk file can be retrieved after closing the disk file and, outside the Multi-User Debugger task, entering the PR_QK command (see the Commands manual for details.)

Determining/Setting the Active Level

The active level is the priority level currently in effect. Directives relating to specific task context are effective only on the active level. When the Debugger is activated by a breakpoint or trace trap, the active level is automatically set. Thereafter, the active level is determined based on the Debugger action in progress; i.e., breakpoint, trace trap, or temporary reference to a different level.

To reference specific task context on another priority level, change the active level by respecifying the Set Level directive (SL) or temporarily designate another level as the active level by specifying the Set Temporary Level directive (TL); in the latter case, the level is considered the temporarily active level. After the desired actions are performed on the temporarily active level, the active level reverts to the level specified in the previous Set Level directive.

Following are guidelines for determining which level is the active level, and methods of setting the active and temporarily active level.

1. The Set Level directive (SL) sets (or changes) the active level. The specified level becomes the default level accessible by the operator terminal or another terminal that is the directive input device.

2. The Set Temporary Level directive (TL) designates a level as the temporarily active level; this permits you to display or alter registers of a level different from the default terminal level without permanently changing the default terminal level. The temporarily active level exists for the duration of one input line. The input line consists of the TL directive plus any other directives addressed to that level.

3. Whenever a break or trace point is processed for a task, the active level is set to the level of that task.

Maintaining a Trace History

When using the Debugger with disk-stored directive lines that execute upon encountering a trap or a breakpoint, a trace history may be maintained on the device specified as user-out.

Also, while at a Debugger breakpoint, the suspended task may be set to run in jump-trace mode (j-mode). In this case, every departure from the current sequence of instructions generates a trace trap.

MULTI-USER DEBUGGER DIRECTIVES

The rest of this section consists of detailed descriptions of the Multi-User Debugger directives, presented in alphabetic order.

The following notational symbols are used to describe the format of Multi-User Debugger directives.

Notational Symbols	Meaning
braces { }	For a single enclosed argument, indicates that the argument is optional. If more than one argument is enclosed by braces in a vertical listing, the braces indicate that a choice is to be made. In this case, optional arguments are identified in the text.
Ellipsis (...)	Indicates the ability to repeat within braces.
Delta (Δ)	Indicates one or more spaces.
Vertical bar (|)	Indicates a choice between two or more arguments.

Note that the use of braces shown above differs from the usage defined in the preface and employed in other sections.

ALL REGISTERS

All Registers

The All Registers directive (AR) prints on the device specified as user-out all registers for the active level. Bound unit breakpoints lie within the loader, not in the task context. As a result, the display of registers at a bound unit breakpoint are not those of the task and can be ignored.

FORMAT:

AR

Assign

The Assign Directive (AS) assigns a specified hexadecimal value to a specified symbol; this directive alters registers of the active level, and defines reserved symbols. Bound unit breakpoints lie within the loader, not in your task context. As a result, the Assign directive on a register is refused by the Multi-User Debugger, if the current level's task is suspended on a bound unit breakpoint.

FORMAT:

AS Δ sym Δ exp { Δ sym Δ exp... }

ARGUMENTS:

sym

A reserved symbol G through Z or a register.

exp

An expression that resolves to a hexadecimal value up to 32 bits. The rightmost 20 bits are used for an address register ($Bn), the program counter ($P), or the bound unit entry point ($E); the rightmost 16 bits are used for all other registers.

Example:

```
AS $R1 -2 X 1408 $B7 X+15
```

-2 is assigned to data register 1, 1408 is assigned to the reserved symbol X, and 141D assigned to base register 7.

CHANGE MEMORY

Change Memory

The Change Memory directive (CH) changes the contents of a single specified memory location, or consecutive locations starting at that location, to specified value(s).

NOTE

This directive changes memory only. To alter register contents, see the Assign (AS) directive.

FORMAT:

CHΔexpΔrexp{Δrexp...}

ARGUMENTS:

exp

First or only location whose contents will be changed.

rexp

Value(s) to be put in memory location(s).

Example 1:

CH 200 4FFF 1716

Put the value 4FFF into location 200 and 1716 into location 201.

Example 2:

CH 100 0/10

Locations 100 to 10F are zero-filled.

Example 3:

CH 2000 0/10 1/10 2/10

This example shows how multiple repeat factors can be used: Locations 2000 to 200F are given a value of zero, locations 2010 to 201F are given a value of 1, and locations 2020 to 202F are filled with 2s.

Clear Abnormal Trap Bit

Clear the abnormal trap bit set in the debugger's indicator word.

This bit is set to request that a special debug breakpoint message be displayed if a task in a group encounters an unexpected (abnormal) 0303xx trap condition. If the bit is not set, the trap information is displayed and the task is terminated.

With the bit set, the trap information is displayed, the task is suspended, and a special breakpoint message appears. These events allow the user to decide whether to continue executing the task (by entering GO) or to terminate the task (by entering TT).

FORMAT:

```
CT
```

CLEAR ALL BOUND UNIT BREAKPOINTS

Clear All Bound Unit Breakpoints

The Clear All Bound Unit Breakpoints directive (CB*) clears all bound unit breakpoints, but not their associated directive lines.

FORMAT:

```
CB*
```

Clear All Quick Breakpoints

The Clear All Quick Breakpoints directive (CQ*) clears all quick breakpoints, but not their associated directive lines.

FORMAT:

```
CQ*
```

CLEAR ALL TRUE BREAKPOINTS

Clear All True Breakpoints

The Clear All True Breakpoints directive (C*) clears all defined true breakpoints, but not their associated directive lines.

FORMAT:

```
C*
```

Clear Bound Unit Breakpoint

The Clear Bound Unit Breakpoint directive (CBn) clears a specified breakpoint for a bound unit, but does not clear the associated directive line.

FORMAT:

CBn

ARGUMENT:

n

Specifies the bound unit breakpoint to be cleared; must be a decimal digit from 0 to 9.

Example:

CB3

Breakpoint number 3 is cleared for the bound unit previously defined by SB3; the associated directive line is not cleared.

CLEAR QUICK BREAKPOINT

Clear Quick Breakpoint

The Clear Quick Breakpoint directive (CQn) clears a specified quick breakpoint, but not the associated directive line.

FORMAT:

```
CQn
```

ARGUMENT:

n

Number of the quick breakpoint; must be a decimal digit from 0 through 9.

Example:

```
CQ3
```

Quick breakpoint number 3 is cleared; the associated directive line is not cleared.

Clear True Breakpoint

The Clear True Breakpoint directive (Cn) clears a specified true breakpoint, but not the associated directive line.

FORMAT:

Cn

ARGUMENT:

n

Number of the true breakpoint; must be from 0 through 31 (decimal).

Example:

C3

True breakpoint number 3 is cleared; the associated directive line is not cleared.

Conditional Execution

The Conditional Execution directive (IF) allows a set of conditions to be tested prior to execution of other Multi-User Debugger directives. The IF directive is intended to be used in a stored breakpoint directive line. It permits breakpoints to be reported without suspending the active level if the specified condition does not exist. When a breakpoint occurs for which an IF directive has been specified, the following actions occur:

- Any directives preceding IF are executed.
- The IF conditions are evaluated, as follows:

  If TRUE, a line in the following format is displayed on the current Debugger output device

  $$\text{"exp } \left\{\text{^}\right\}\left\{\begin{matrix}<\\=\\>\end{matrix}\right\}\left\{\text{,}\right\}\text{hhhh..."}$$

  and any directives following IF are executed. If a GO directive does not follow, the active level is suspended.

  If FALSE, no display occurs, and the directives following IF are not executed. The active level continues processing.

FORMAT:

$$\text{IF exp } \left\{\text{^}\right\}\left\{\begin{matrix}<\\=\\>\end{matrix}\right\}\left\{\text{,}\right\}\text{hhhh...;}$$

ARGUMENTS:

exp

Hexadecimal memory address of a byte string argument. This must specify an address; $Rn (where $0 \leq n \leq 7$) cannot be used for exp. Since no check for this error is performed, however, if you use $Rn, results are unpredictable.

{^}{<=>}

Specifies the condition to be tested when comparing the memory byte string value to the test parameter. optionally specifies logical negation; i.e., not less than, not equal, not greater than.

{'}

Indicates that the argument is r.ght-byte aligned.

hhhh...

The test parameter, expressed in ASCII as a string of pairs of hexadecimal digits; each pair represents one byte. The test parameter may not be an assigned symbol (see the Assign (AS) directive). The length of the parameter is limited by the maximum size of a Multi-User Debugger stored directive (127 bytes). The parameter's ASCII value must consist of pairs of hexadecimal values. If an odd number of hexadecimal values are specified, a command error is reported when the directive is executed and the task remains suspended to allow for correction. If the IF directive is embedded in a Quick Breakpoint directive line, this error condition is a false state and the rest of the directive line is ignored and the task will continue. The IF directive terminator must be a semicolon (;).

Example:

Assume that true breakpoint 2, as defined below, is encountered, and that $B7 points to memory location 555F:

```
S2 135E (IF 1000^>,3E;IF $B7=42D1;DP $B7/100;GO)
```

Two conditions must be true before the Dump (DP) directive is executed:

1. The rightmost byte at memory location 1000 must be less than or equal to 3E.

2. The byte string found at memory location 555F must be equal to 42D1.

If both conditions are met, the dump is executed, and the active level continues in response to the GO directive. If either condition is not satisfied, the dump does not occur, and the active level continues without suspension.

NOTE

The IF directive can be entered from the terminal, in which case its action corresponds to its entry in a stored directive line. However, using the IF directive from the terminal is of limited usefulness, since the conditions to be tested can be checked by using other directives (e.g., DH).

Define Directive Line

The Define directive (Dn) defines a specified directive line for future use and associates that line with a specified number. The directive line is stored on the user-defined work file and can be referred to by specifying in an Execute (En) directive the number with which it was associated. The entire Define directive may comprise a maximum of 126 characters.

When you reuse a disk that has predefined directive lines from a previous execution, the lines may be referred to without redefining them. (See "Set True Breakpoint Directive (Sn).") This prevents complex predefined directive lines from being respecified each time the system is reloaded for debugging the same problem.

FORMAT:

DnΔ(directive line)

ARGUMENTS:

n

Number with which the specified directive line is associated; must be from 0 through 9.

(directive line)

One or more directives stored for future use.

Example 1:

D3 (CH 100 0)

Associate the number 3 with the directive within the parentheses. Hereafter, each time the directive E3 (see "Execute Directive (En)" below) is executed, the parenthetical directive is executed and location 100 is zero-filled.

Example 2:

D4 ( )

By storing a null directive, deactivate a previously defined directive line 4 which is no longer required.

DEFINE TRACE

Define Trace

The Define Trace directive (DT) associates the directive line within the parentheses with the occurrence of a jump trace trap or a BRK instruction not already defined as a breakpoint. The specified directive line is stored in the user-defined work file for future use. The entire Define Trace directive may comprise a maximum of 126 characters.

When you reuse a disk file that has predefined directive lines from a previous execution, the lines may be referred to without redefining them. (See "Set True Breakpoint Directive (Sn).")

FORMAT:

DTΔ(directive line)

ARGUMENT:

(directive line)

One or more directives stored for future use.

Example 1:

DT (AR)

All registers are displayed each time a trace trap occurs. (See "All Registers Directive (AR).")

Example 2:

DT ( )

Cancel usages of the predefined trace directive line.

Display Memory

The Display Memory directive (DH) displays one or more specified memory location(s) in hexadecimal notation either on the terminal or on another specified device.

FORMAT:

DHΔrexp{Δrexp...}

ARGUMENT:

rexp

Location(s) whose contents are displayed. A minimum of one location may be displayed.

Example 1:

```
DH 200
```

Display the contents of location 200.

Example 2:

```
DH 200/100
```

Display the contents of locations 200 to 2FF.

DUMP MEMORY

Dump Memory

The Dump Memory directive (DP) prints on the terminal or another specified device an area of memory starting at a specified location. The printout comprises a minimum of eight locations and is in hexadecimal and ASCII notations.

NOTE

Up to 32K words of memory can be dumped in response to a single DP directive. Dumps of more than 32K must be performed as separate operations.

FORMAT:

DPΔrexp{Δrexp...}

ARGUMENT:

rexp

Memory location(s) whose contents are displayed. The display is always in a multiple of eight locations.

Example 1:

```
DP 200
```

Display (at the current user-out device) one line of memory in both hexadecimal and ASCII, starting at location 200.

Example 2:

```
DP 80/3C 200/240
```

Display the contents of locations 80 to BF and 200 to 43F on the current user-out device. Although the repeat expression of 3C was specified in the directive, the display is through location BF because displays are always in multiples of eight locations.

End Trace

The End Trace directive (ET) disables the j-mode trace (see the Start j-mode Trace directive (ST)) for a specific task on the next trap.

FORMAT:

```
ETΔlvl
```

ARGUMENT:

lvl

The level, as previously specified by the last ST directive. lvl is preceded by one space. The trace must first have been enabled using the Start j-mode Trace directive (ST).

ESCAPE

Escape

The Escape directive (E) passes the rest of the input buffer to the command processor for processing. Debugger directives can precede the portion of the input buffer to be passed to the command processor, if they are separated by semicolons (;). They cannot, however, follow commands passed to the command processor. Once an Escape directive has been encountered, the rest of the input line is interpreted by the command processor. This allows multiple commands to be passed to the command processor using only one escape directive.

FORMAT:

EΔexp$\left\{;\text{exp}\right\}$

ARGUMENT:

exp

Any command.

Example:

```
E TIME
```

Return the time.

NOTE

Do not use the Escape directive to invoke the bound units that you intend to debug. The Multi-User Debugger must be terminated (see the Sleep directive (SP) for details) before invoking a bound unit containing breakpoints.

Execute

The Execute directive (En) retrieves and executes a specified predefined directive line. This directive may not be embedded in Define directive (Dn) lines; it is permitted in Set True Breakpoint (Sn), Define Trace (DT), and Set Bound Unit (SBn) Breakpoint lines.

FORMAT:

En

ARGUMENT:

n

Number of the line to be executed; must be from 0 through 9.

Example 1:

```
D3 (CH 100 0)
E3
```

The directive E3 retrieves and executes line 3, previously defined in the Define directive as CH 100 0.

Example 2:

```
D3 (CH 100 0)
S1 100 (E3)
```

The Execute directive (E3) is embedded in a Set True Breakpoint directive line. The Execute directive retrieves and executes line 3, previously defined in the Define directive as CH 100 0, whenever true breakpoint 1 is encountered.

FILE OUT

File Out

The File Out directive (FO) redirects output from the current user-out file to the device specified by the pathname argument. This directive allows messages that result when a true or bound unit breakpoint or other condition occurs to be sent to a device other than the user-out file. It has no effect on input to the program.

FORMAT:

FO {path}

ARGUMENT:

path

The pathname of the device to which output for the group is directed. If path is omitted, user-out defaults to the group's original user-out file.

Example:

FO !LPT00

Output is redirected from the current user-out file to a line printer.

Get Quick Memory

The Get Quick Memory directive (MQ) reserves the requested amount of memory for storing (in memory buffers) the output from execution of a Quick Breakpoint directive line.

FORMAT:

MQΔ {-BS exp} Δ {-RS exp} Δ {-NB exp}

ARGUMENTS:

-BS exp

Size of buffer specified in words (hexadecimal).

Default: 800

-RS exp

Size of record specified in words (hexadecimal).

Default: 100

-NB exp

Number of buffers requested; must be two or more.

Default: 2

NOTES

1. To use quick breakpoints, enter this directive first after invoking the Multi-User Debugger in the $S task group.

2. Each buffer must contain at least two records. The first record of each buffer contains the information needed by the Multi-User Debugger as listed below.

Offset	Number of Words	Definition
0	4	File system information
4	1	Quick file identifier
5	2	Pointer to next buffer in chain
7	2	Pointer to next available record in buffer
9	1	Maximum record size
10	1	Number of records still available in buffer
11	1	Number of records not completed in buffer
12	1	Number of records for reset
13	1	Current buffer number (n+1)
14	1	Number of buffers in the memory block
15	1	Indicators word X'1' buffer in use X'2' buffer full X'4' buffer ready for reuse X'8' buffer information lost X'10' last buffer to be written to disk X'20' disk file has cycled

The minimum size for each record is 19 words.

Example:

```
MQ -RS 80
```

Request memory. The default values for buffer size and the number of buffers requested are used. Each record is 80 words long.

GO

The GO directive resumes execution on the current active level after a breakpoint and can optionally specify a limit-to-pause counter value which applies only to j-mode trace traps (see the Start j-mode Trace directive (ST)).

FORMAT:

GO $\left\{ \Delta LLLL \right\}$

ARGUMENT:

LLLL

Optionally, an ASCII expression of 1 to 4 hexadecimal digits greater than zero. The ASCII expression is preceded by one space.

Default: 1

Example:

```
S0 100 (DH 200/10;GO)
```

The task encountering true breakpoint 0 traps; the Associated directive line is executed by the Multi-User Debugger and the last directive of the directive line (GO) reactivates the task.

LIST ALL BOUND UNIT BREAKPOINTS

List All Bound Unit Breakpoints

The List All Bound Unit Breakpoints (LB*) directive displays all bound unit breakpoints and their associated directive lines if the work file is open. If the work file is not open, only defined bound unit breakpoints are listed. If the work file is open, the listing contains all bound unit breakpoints and all associated directive lines. It is possible to list a bound unit breakpoint with no corresponding directive line or a directive line with no defined bound unit breakpoint. However, if neither a bound unit breakpoint nor a directive line is defined for a particular bound unit breakpoint number, that bound unit breakpoint number does not appear in the list.

FORMAT:

```
LB*
```

Sample Listing:

```
BU0        (S0   $E;GO)
BU2 LWD ( )
```

The work file is open and bound unit 0 has a directive line but no defined breakpoint; bound units 1 and 3 through 9 have neither defined breakpoints nor directive lines; and bound unit 2 has only a defined breakpoint.

NOTE

Ten bound unit breakpoints (one per bound unit; 0 through 9) can be set. See the Set Bound Unit Breakpoint directive (SBn) description below.

List All Quick Breakpoints

The List All Quick Breakpoints directive (LQ*) displays all quick breakpoints and their associated directive lines. You can print a directive line without an associated quick breakpoint (e.g., if the quick breakpoint has been previously cleared). If neither a quick breakpoint nor a quick breakpoint directive line is defined for a particular quick breakpoint number, that breakpoint does not appear in the list.

FORMAT:

```
LQ*
```

Sample Listing:

```
QUICK BREAKPOINTS

1 LOC = ABCD INST = 0F03 (GO)
3                        (DP $P;AR;GO)
```

Directive lines are defined for quick breakpoints 1 and 3, although breakpoint 3 is not currently set. Quick breakpoints 0 and 2 through 9 have neither a defined breakpoint nor a directive line.

NOTE

Ten quick breakpoints (0 through 9) may be set. See the Set Quick Breakpoint directive (SQn) description below.

LIST ALL TRUE BREAKPOINTS

List All True Breakpoints

The List All True Breakpoints directive (L*) lists all currently defined true breakpoints, their location in memory, the instruction which was replaced, and their associated directive lines. If the work file is not open, the list consists of the locations of the defined true breakpoints and the instruction being replaced. If the work file is open, all defined true breakpoints and all associated directive lines are listed. It is possible to list a true breakpoint without an associated directive line, or a directive line without an associated true breakpoint. However, if neither a true breakpoint nor a directive line is defined for a particular true breakpoint number, that breakpoint number does not appear in the list.

FORMAT:

```
L*
```

Sample Listing:

```
TRUE BREAKPOINTS
1 LOC = 00ABCD INST = 0F02 ( )
3                          (AR;DP $P;GO)
```

True breakpoint 1 is listed with no directive line and true breakpoint 3 has only a defined directive line. True breakpoints 0 and 2 through 31 have neither a defined true breakpoint nor directive line.

NOTE

32 true breakpoints (0 through 31) may be set. See the Set True Breakpoint (Sn) directive description below.

List Bound Unit Breakpoint Directive

The List Bound Unit Breakpoint directive (LBn) displays the stored directive line associated with a specified bound unit breakpoint.

FORMAT:

LBn

ARGUMENT:

n

Number of the bound unit breakpoint for which the directive line is to be listed; must be from 0 through 9.

Example:

LB3

List the directive line associated with bound unit breakpoint 3.

LIST QUICK BREAKPOINT

List Quick Breakpoint

The List Quick Breakpoint directive (LQn) displays a particular quick breakpoint number set by a Set Quick Breakpoint (SQn) directive, and its associated directive line. You can print a directive line without an associated quick breakpoint (e.g., if the quick breakpoint had been previously cleared).

FORMAT:

LQn

ARGUMENT:

n

Number of the quick breakpoint whose directive line is listed; must be a decimal digit from 0 through 9.

Example:

LQ2

Display the directive line associated with quick breakpoint 2.

List True Breakpoint

The List True Breakpoint directive (Ln) displays a particular true breakpoint number set by a Set True Breakpoint (Sn) directive, and its associated directive line.

FORMAT:

Ln

ARGUMENT:

n

Number of true breakpoint whose directive line is listed; can be 0 through 31 (decimal).

Example:

L2

Display the directive line of true breakpoint 2.

MODE

Mode

Change the current mode of the debugger--from numeric to symbolic or vice-versa.

FORMAT:

```
MODE  NUM[ERIC]
      SYM[BOLIC]
```

DESCRIPTION:

The debugging mode is set as specified.

Example:

```
MODE  SYM
```

The debugger is currently in numeric mode; the directive changes the mode to symbolic.

NOTE

A detailed description of the symbolic mode directives and their use is found in the Application Developer's Guide.

Print

The Print directive (Pn) prints specified lines predefined by Dn directives. Use the Print All directive (P*) to print all predefined lines.

FORMAT:

Pn

ARGUMENT:

n

Number of the line to be printed; can be 0 through 9.

PRINT ALL

Print All

The Print All directive (P*) prints all lines predefined by Dn directives. Use the Print directive (Pn) to print only specified predefined lines.

FORMAT:

p*

Print Header Line

The Print Header Line directive (Hn) prints a specified header line starting at the head of form or after a specified number of lines are skipped. The main uses of the Print Header Line directive are to document printed information related to breakpoint or trace trap debugging, and to annotate a line printer memory dump.

FORMAT:

HnΔ(headerΔ)

ARGUMENTS:

n

Number of lines skipped before header line is printed; can be 1 through 9, or 0. 0 causes header to be printed at head of form.

(header)

Any ASCII characters and/or expressions; each expression must be preceded by a percent (%) sign. If a percent sign is to be printed, two percent signs must be used (%%). Left and right parentheses must be balanced within header lines.

Example:

```
H0 (DUMP OF BREAKPOINT FOR LEVEL %$S )
```

Document dumps. As soon as a carriage return is typed, the above header is printed at the top of a new page.

PRINT HEXADECIMAL VALUE

Print Hexadecimal Value

The Print Hexadecimal Value directive (VH) prints, in hexadecimal, the value of each specified expression.

FORMAT:

VHΔexp{Δexp}

ARGUMENT:

exp

Expression whose value is displayed.

Example:

```
VH .+100-M
```

Display the result of the computation defined by the last referenced memory location plus 100 (hexadecimal) minus the value assigned to the temporary symbol M.

Print Quick Memory Pointer

The Print Quick Memory Pointer directive (PQ) prints the hexadecimal address of the start of quick memory.

FORMAT:

PQ

PRINT TRACE

Print Trace

The Print Trace directive (PT) prints a defined trace directive line.

FORMAT:

PT

Quit

The Quit directive (QT) clears all breakpoints, closes all Debugger work files, and disables the Debugger trap handler before aborting the Multi-User Debugger task.

FORMAT:

```
QT
```

RESET FILE

Reset File

The Reset File directive (RF) closes files and prohibits execution of directives that refer to user-defined files.

FORMAT:

RF {DB / QK}

ARGUMENTS:

DB
QK

DB - Close the Debugger work file and prohibit execution of the P*, Pn, PT, Sn, Dn, DT, En, SBn, Ln, and LBn directives. These directives may not be entered until another specify file directive (SF) is issued to open a new work file.

QK - Close the quick disk file and prohibit the quick memory buffers from being written to the file. If no quick breakpoints are currently set prior to the issuing of an RF directive, the following occurs:

(1) The current buffer is marked "last" used and full.

(2) The quick disk file is closed after the "last" buffer is written to the disk file.

(3) The writer task is terminated.

If there are quick breakpoints still set, steps (1) and (3) are done, but step (2) cannot be guaranteed to write all the buffers to the disk before the file is closed.

Return Quick Memory

The Return Quick Memory directive (RQ) causes (1) the quick disk file to be closed after all memory buffers used have been written to it, (2) the asynchronous writer task to be terminated, and (3) memory to be returned to your pool.

NOTE

Both the quick memory and that memory necessary for quick breakpoint processing are returned to your pool.

FORMAT:

```
RQ
```

Set Bound Unit Breakpoint

The Set Bound Unit Breakpoint directive (SBn) sets a numbered breakpoint for a specified bound unit or overlay. A given bound unit (BU) breakpoint refers to either roots or to overlays, or to both. When the bound unit breakpoint is encountered, a message informs you where the specified bound unit or overlay has been loaded into memory, so that you can then set true breakpoints at specified locations in the program. Because a bound unit is loaded at the time the task associated with it is created, the level number displayed when a BU breakpoint occurs is not necessarily the one used when requests for that task are later executed.

The message format is:

```
*BU n $SL=00xx $E=00xxxx + 00xx
```

n

Number of bound unit breakpoint; can be 0 through 9.

$SL=00xx

Specifies priority level.

$E=00xxxx + 00xx

Represents the bound unit base address plus entry point offset as defined by the bound unit or by the caller. Used in place of $P associated with true breakpoints.

FORMAT:

```
        ⎧ bound-unit-name                    ⎫
        ⎪ bound-unit-name/overlay-number     ⎪
SBnΔ    ⎨ bound-unit-name/*                  ⎬ Δ(directive line)
        ⎪ *                                  ⎪
        ⎪ */overlay-number                   ⎪
        ⎩ */*                                ⎭
```

ARGUMENTS:

n

Bound unit breakpoint number; can be from 0 to 9.

bound-unit-name

Name of the bound unit to which the breakpoint applies; up to six ASCII characters (first six characters of the bound unit name).

overlay-number

Hexadecimal number of the bound unit overlay.

*

Stands for "all" roots or "all" overlays, depending on context.

(directive line)

Directives to be executed when the bound unit/overlay is loaded.

Example:

```
SB6 S00Z/A (IF 3D02=5354;VH M-2;GO)
```

Sets breakpoint 6 for overlay number A (hexadecimal) of the bound unit named S00Z. The directive line specifies that if the condition indicated is true (location 3D02 equals 5354), then the value of M minus 2 is displayed. When overlay A is loaded into memory, its location is displayed at the terminal, and the directive line associated with bound unit breakpoint 6 is executed.

SET LEVEL

Set Level

The Set Level directive (SL) sets the active priority level to a specified value. This level remains in effect until another SL directive is issued or a new breakpoint is encountered. The level may be temporarily changed via the Set Temporary Level directive (TL) (see below).

FORMAT:

SLΔexp

ARGUMENT:

exp

Number of active priority level in hexadecimal notation.

Default: 0

Example 1:

```
SL C
```

Sets the active priority level to 12 (decimal). If the AR directive is entered after the above SL directive, the registers on level 12 are displayed.

Example 2:

Designate the active level, permanently change it, and temporarily change it.

```
SL C     The active level is 12 (decimal)
  .
  .
  .
SL A     The active level is 10 (decimal)
  .
  .
  .
TL B;AR  The active level temporarily is 11 (decimal)
```

After the desired action(s) are performed, the active level reverts to level 10 (the level specified in the last SL directive).

Set Quick Breakpoint

The Set Quick Breakpoint directive (SQn) sets a numbered quick breakpoint at a specified location. When the breakpoint is encountered, the stored specified directive line is executed and the Debugger task continues. A message and any information requested by the directive line are written to quick memory and, optionally, to a quick disk file.

The entire Set Quick Breakpoint directive may comprise a maximum of 124 characters.

If there is a preexisting directive line associated with a given quick breakpoint and that directive line is no longer applicable, clear the line by designating empty parentheses ( ) when resetting the quick breakpoint.

The message format is:

($S)QBn Group Id TCB ptr Level

n

Quick breakpoint number; must be 0 through 9.

Group Id

Name of the group under which the task being debugged is running.

TCB ptr

Location of the task control block of the task being debugged.

Level

Priority level of the task being debugged.

NOTES

1. A quick breakpoint cannot be set in any of the following instructions: input/output, generic (BRK), scientific, invalid instruction, LEV, ENT, LNJ, JMP, STS, or any instruction with an invalid address symbol.

2. A GO directive should be the last directive specified in a quick breakpoint directive line. A GO directive embedded in an SQn directive allows task execution to proceed after the desired operation has been performed. The Multi-User Debugger appends a GO directive to the directive line.

3. If the NR argument is specified, there is no evidence that the quick breakpoint has been encountered. If the directive line contains an IF directive and the condition specified is true, any requested information will be stored in the memory buffer.

FORMAT:

SQnΔexpΔ{NR}Δ(directive line)

ARGUMENTS:

n

Number of the quick breakpoint; can be 0 through 9.

exp

Location at which the quick breakpoint occurs.

{NR}

Quick breakpoint output is not stored in the memory buffer.

(directive line)

Directives that are executed when the quick breakpoit is reached. The directives allowed in a quick breakpoint directive line are: AR, AS, CH, DH, GO, HS, IF, and VH. The GO directive should only appear as the last entry in the directive line; if omitted, the GO directive will be appended.

If GO is the only directive specified in the directive line, only the message described above is stored in the memory buffer.

Example:

```
SQ1 1D8B NR (IF 1000<,3E;AR;GO)
```

A quick breakpoint numbered "1" is set at location 1D8B. If the condition specified in the directive line is false, no information is stored in the memory buffer. If the condition is true, the breakpoint message and the contents of the active registers are stored in the memory buffer.

SET TEMPORARY LEVEL

Set Temporary Level

The Set Temporary Level directive (TL) sets the active priority level to a temporary, specified value. The level specified in the TL directive remains in effect until an SL or another TL directive is issued, or until the end of the directive line. If the end of the line is reached before another SL or TL directive is encountered, the value specified in the last SL directive becomes the active priority level. See the Set Level (SL) directive above.

FORMAT:

```
TLΔexp
```

ARGUMENT:

exp

Value designating the temporarily active priority level.

Example:

```
SL    20
TL    A;AR
TL    B;AR
```

The first TL directive designates level 10 as the temporarily active priority level so that all registers on that level can be displayed via the subsequent AR directive.

The second TL directive designates level 11 as the temporarily active priority level so that all registers on that level can be displayed via the subsequent AR directive.

After the last TL directive is executed, the active level is 32 (decimal): the level specified in the last set level directive (SL).

Set True Breakpoint

The Set True Breakpoint directive (Sn) sets a numbered true breakpoint at a specified location. When the true breakpoint is encountered, the stored specified directive line, if any, is executed; otherwise, there is a typeout indicating the contents of the location counter and the active priority level, and the task execution is suspended. The Set File directive (SF) is a precondition for directive line execution. The entire Set True Breakpoint directive may comprise a maximum of 126 characters.

If there is a preexisting directive line associated with a given true breakpoint and that directive line is no longer applicable, clear the line by designating empty parentheses ( ) when resetting the true breakpoint.

The message format is:

```
($H) BPn $P=00xxxx $SL=00xx
```

$P=00xxxx

Location counter

$SL=00xx

Priority level

NOTES

1. If a true breakpoint is set in any of the following types of instructions, the true breakpoint must be cleared (Cn directive) before continuing execution (GO directive): input/output, generic (BRK), scientific, LEV, invalid instruction, or instruction with an invalid address syllable. To avoid this restriction, clear the existing true breakpoint and then reset it in a subsequent Set True Breakpoint directive.

2. A GO directive embedded in an Sn directive line allows task execution to proceed after the desired operations have been performed, without further operator intervention.

FORMAT:

Sn∆exp{∆(directive line)}

ARGUMENTS:

n

Number of true breakpoint; can be 0 through 31 (decimal).

exp

Location at which true breakpoint occurs:

(directive line)

Directives that are executed when true breakpoint is encountered.

Example 1:

```
S0 100 (DH 200/10;GO)
```

Display locations 200 to 20F when location 100 is reached, then proceed from breakpoint.

Example 2:

```
S0 100 ( )
```

Cancel any line previously associated with true breakpoint 0.

Example 3:

```
S0 1000 (AR;C0;GO)
S1 1003 (S0 1000;GO)
```

The first directive line sets true breakpoint number 0 at location 1000, prints all registers on the active level, clears true breakpoint number 0 because the instruction at location 1000 is restricted (see Note 1 above), then proceeds from the breakpoint.

The second directive line sets true breakpoint number 1 at location 1003 and then reestablishes true breakpoint 0 at location 1000; the second true breakpoint line causes no visible action except the printing of the breakpoint message.

Sleep

The Sleep directive (SP) temporarily suspends the execution of the Multi-User Debugger and returns control to the command processor.

FORMAT:

```
SP
```

SPECIFY FILE

Specify File

The Specify File directive (SF) identifies the relative or full pathname of the Multi-User Debugger file to be opened. Since the function of the SF directive is to locate the file, first execute this directive; otherwise, an error message appears as soon as a directive requiring the file is used. When using quick breakpoints, the first directive entered after invoking the Multi-User Debugger should be the Get Quick Memory directive; the Specify File directive, in this case, should be the second directive entered.

FORMAT:

$$SF\,\Delta \begin{Bmatrix} path \\ path\ -CYCLE \\ -CYCLE \end{Bmatrix}$$

ARGUMENTS:

path

Relative or full pathname of the file to be opened; relative pathname can be 1 to 12 characters in length. All Multi-User Debugger work files must end with the suffix .DB; all Multi-User Debugger quick disk files must end with the suffix .QK.

-CYCLE

Used only with quick disk files. At end of file, returns the Debugger writer task, which enters debug information into the file, to the beginning of the quick disk file. If -CYCLE is not specified, the quick disk file is closed at end of file even if there was more data to be written.

NOTE

If you did not initially specify the -CYCLE argument and desire to do so later in the Debugger session, enter

SF -CYCLE

at any point later in the Debugger directive sequence before end of file; this causes the Debugger writer task to return to the beginning of the quick disk file when it reaches end of file.

Example 1:

```
SF GLASS.DB
```

Work file GLASS.DB is opened.

Example 2:

```
SF GLASS.QK -CYCLE
```

Quick disk file GLASS.QK is opened and, when end of file is reached, the Debugger writer task returns to the beginning of the quick disk file to continue entering input into the file.

Example 3:

```
SF GLASS.QK
    .
    .
    .
SF -CYCLE
```

Quick disk file GLASS.QK is opened. Later in the program, force the writer task to return to the beginning of the quick disk file to complete the writing task.

NOTES

1. If the .QK or .DB suffix is not specified in the SF directive line for a work file, it is assumed a work file is being requested and .DB is appended before the file is opened.

2. If the specified work file does not exist, it is created and opened with exclusive read/write access when the SF directive is entered. Only one user has access to a Debugger work file at any given time.

3. If a simple pathname is entered, the system looks only in the current working directory for the specified work file.

4. You have the option of changing work files by entering a new SF directive, thereby closing the currently active file and opening a new file in its place.

5. The quick disk file must have been previously created outside the Multi-User Debugger task using the Create File (CR) command. The pathname supplied must include the suffix .QK. The argument values supplied must agree with those used in the Get Quick Memory (MQ) directive. In the following example, the argument values are the default values of the MQ directive.

   CR filename.QK -REL -CISZ 4096 -SZ n -LRSZ 512

   where n must be greater than or equal to 2.

6. The .QK suffix must be specified in the SF directive line for a quick disk file; otherwise, a default value of .DB is appended and a work file is opened.

7. The quick disk file can only be opened when running the Multi-User Debugger from the system ($S) group. Only one quick disk file can be opened at any given time. When the SF directive is specified, this quick disk file has exclusive write access.

8. Opening a quick disk file spawns the Debugger writer task. The writer task terminates when the quick disk file is closed. This task runs on level 62.

9. You have the option of changing the quick disk file by entering a reset file (RF) directive, thereby closing the currently active file. Then enter a new specify file (SF) directive to open the new file.

Start j-mode Trace

The Start j-mode Trace directive (ST) sets the given task's M1 register j-bit on. As a result, any departure from the current processing sequence causes a trap. The Multi-User Debugger treats the trap as a "trace trap." The following points apply:

- j-mode trace can be started only for a task which is currently suspended due to a true breakpoint.
- The "Start j-mode Trace" directive is refused if the task is suspended due to a bound unit breakpoint.
- j-mode processing is specific to a given task and is shut off or restored at the monitor call interfaces.
- When a task is running in j-mode, the Multi-User Debugger's handling of successive traps is governed by the "limit-to-pause" counter of the GO directive.
- Limit-to-pause has a default value of 1, but may be set to an arbitrary value via the GO directive. The Multi-User Debugger decrements the limit-to-pause once for each occurrence of a trace trap. When limit-to-pause assumes the value zero, the trapped task is suspended to permit operator action and a TRACE PAUSE message is issued. When the task is reactivated (GO [LLLL]) the limit-to-pause is reset to the default value or to a user-specified value.

FORMAT:

STΔlvl

ARGUMENT:

lvl

Active level of the task in question.

TURN ON ABNORMAL TRAP BIT

Turn On Abnormal Trap Bit

Turn on the abnormal trap bit in the debugger's indicator word.

This bit is set to request that a special breakpoint message be displayed if a task in the specified group encounters an unexpected (abnormal) 0303xx trap condition. With the bit set, the trap information is displayed, the task is suspended, and the special breakpoint message is displayed. At this time, debug has control of the group, allowing the user to determine what caused the trap. The user can then decide to continue from the trap (by entering GO) or to terminate the task (by entering TT).

This bit is automatically set when debug is first invoked in a group. It can be turned off at any time by typing the Clear Abnormal Trap Bit (CT) directive.

The format of the abnormal breakpoint message is:

```
BP TP $SL=00XX $P=00XXXX
```

where:

$P points to the next location in memory following the trapped instruction.

FORMAT:

TB

Terminate the Trapped Task

Terminate the request previously entered against the trapped task.

A user who decides that the task being debugged has been sufficiently examined can terminate the task by this directive. The TT directive can be executed only for a task suspended by a true or special trap breakpoint.

NOTE

If TT terminates a task that has abnormally trapped and is now suspended on the special breakpoint, there will be no evidence of that task left in the task group. Normally, the trapped task's TCB and associated TSAs are left in the group for later analysis of a dump.

FORMAT:

```
TT
```

SAMPLE MULTI-USER DEBUGGER SESSIONS

Three sample debugging sessions are shown below to illustrate some of the directives and procedures described earlier in this section. The second and third debugging sessions illustrate primarily the use of quick breakpoints.

Sample Session 1

The bound unit being debugged is TEST, listed in Figure 18-1. TEST takes as an argument a number in the range 0 through 2. The function of TEST is to write to user-out one of three numbered messages; the message number should correspond to the number entered as the argument.

The debugging session is shown in Figure 18-2.

```
     TITLE TEST          07/22/80  1557.8 edt tue        HRS ASSEMBLER 6.02  -SLIC  PAGE  0001
000001                                       TITLE     test,'80072200'      Debug Test Program
000002                                       *
000003                                       *         COPYRIGHT, (C), 1980, HONEYWELL INFORMATION SYSTEMS, INC.
000004                                       *
000005          0000                         start     equ       $
000006  0000    F877                         ldr       $r7,@$b7             arg count from FRB
000007  0001    9870 1702                    ldr       $r1,=X'1702'         error status
000008  0003    7702                         cmv       $r7,=2               two or more args?
000009  0004    0216                         bl        >error               br if no
000010  0005    B067                         cmn       +$b7                 move passed first arg
000011  0006    9C87                         ldb       $b1,$b7              ptr to second argument
000012  0007    9871                         ldr       $r1,@$b1             arg size
000013  0008    8A01                         inc       =$r1
000014  0009    1001                         sol       $r1,1                convert to words
000015                                       *
000016          000A                         arg       equ       $
000017  000A    AC71                         ldb       $r2,$b1              get second arg
000018  000B    2100                         adv       $r2,-x'30'           drop ascii bits
000019  000C    ABC0 0017                    lab       $b2,msgdsp
000020  000E    C822                         ldr       $r6,$b2,$r2          length of arg
000021  000F    A006                         ldb       $r2,=$r6             get length
000022  0010    A570 00FF                    and       $r2,=x'00FF'         drop sign extension
000023  0012    CUC0 0015                    lab       $b4,msg              ptr to msg table
000024  0014    CCA4                         ldb       $b4,$b4,$r2          move to desired msg
000025  0015    6048                         sor       $r6,8                shift length to right byte
000026  0016    7C00                         ldv       $r7,0
000027  0017    0001                         mct
000028  0018    0801                         dc        x'0801'              call USER_OUT
000029  0019    1907                         bez       $r1,>done            br if no error
000030  001A    0870 0080                    error     ldr       $r3,=x'80'         component code
000031  001C    F851                         ldr       $r7,=$r1             error code
000032  001D    6C00                         ldv       $r6,0
000033  001E    0001                         mct
000034  001F    0F00                         dc        x'0F00'              call REPORT ERROR
000035                                       *
000036          0020                         done      equ       $
000037  0020    1C00                         ldv       $r1,0
000038  0021    2C00                         ldv       $r2,0
000039  0022    0001                         mct
000040  0023    0103                         ic        x'0103'              call TERMINATE TASK
000041                                       *
000042          0024                         msgdsp    equ       $
000043  0024    0606                         dc        msg0-msgdsp*256+(msg0_l-1)
000044  0025    0F00                         dc        msg1-msgdsp*256+(msg1_l-1)
000045  0026    0614                         dc        msg2-msgdsp*256+(msg2_l-1)
```

Figure 18-1. Sample Program TEST

```
                TITLE TEST              07/22/80  1557.8 edt tue        HRS ASSEMBLER 6.02  -SLIC  PAGE  0002

     000046  0027  0910                                      dc        msg3-msgdsp+256*(msg3_l-1)
     000047        0028                        msg           equ       $
     000048  0028  4140 4553 5341             msg0          dc        'AMESSAGE ZERO '
                    4745 205A 4552
                    4F20
     000049        0007                        msg0_l        equ       $-msg0
     000050  002F  4154 4553 5420             msg1          dc        'ATEST MESSAGE ONE '
                    4D45 5353 4147
                    4520 4F4E 4520
     000051        0009                        msg1_l        equ       $-msg1              message length
     000052  0038  4140 4553 5341             msg2          dc        'AMESSAGE TWO '
                    4745 2054 5746
                    2020
     000053        0007                        msg2_l        equ       $-msg2
     000054  0034  4154 4553 5420             msg3          dc        'ATEST MESSAGE THREE '
                    4D45 5353 4147
                    4520 5448 5245
                    4520
     000055        000A                        msg3_l        equ       $-msg3
     000056                                    **
     000057  0069  0000                                      end       test.start
0000  ERR COUNT
```

Figure 18-1 (cont). Sample Program TEST

```
   ($H)RDY:
   TEST 1
   ($H),/D
   ($H)RDY:

(1)DEBUG
   ($H)DEBUG-R210-07/18/1310
(2)SB1 TEST
(3)LB*
   ($H) BU1 TEST
(4)SP
   ($H)RDY:
(5)TEST 1
   ($H) *BU 1 $SL=001B $E=00FDDA + 0000  DATA+000000
(6)AS X $E
(7)VH X
   ($H) X=00FDDA
(8)DP X/10
   ($H)
   ($H) 00FDDA/ F877 9870 1702 7D02 0216 8DE7 9C87 9871  .w.p..}........8
   ($H) 00FDE2/ BAD1 1001 A081 2ED0 ABC0 0017 E822 A0D6  ..............".."
(9)S1 X
(10)L*
   ($H) TRUE BREAKPOINTS
   ($H)   1  LOC=00FDDA  INST=F877
(11)DP X
   ($H)
   ($H) 00FDA/ 0002 9870 1702 7D02 0216 8DF7 9C87 9871  ...p..}.:.......q
(12)GO
   ($H) *BP 1 $SL=001C $P=00FDDA
(13)AR
   ($H) $R1=0000 $R2=0000 $R3=0000 $R4=0000 $R5=0000 $R6=0000
   ($H) $R7=0000 $B1=000000 $B2=00FFA6 $B3=000000 $B4=00FFA2
   ($H) $B5=001194 $B6=00FDDA $B7=00FFA6 $P=00FDDA $I=0000 $S=4010
(14)DP $B7/10
   ($H)
   ($H) 00FFA6/ 0002 FFAA FFAE 0000 0004 5445 5354 2020  ..........TEST
   ($H) 00FFAE/ 0001 3120 0102 FFA2 FD82 FD42 0000 8002  ..1........B....
(15)S2 X+A
(16)GO
   ($H) *BP 2 $SL=001C $P=00FDE4
(17)AR
   ($H) $R1=0004 $R2=0000 $R3=0000 $R4=0000 $R5=0000 $R6=0000
   ($H) $R7=0002 $B1=00FFAF $B2=00FFA6 $B3=000000 $B4=00FFA2
   ($H) $B5=001194 $B6=00FDDA $B7=00FFA8 $P=00FDE4 $I=0004 $S=401C
(18)AS $R1 1
(19)DP $B1
   ($H)
   ($H) 00FFAF/ 3120 0102 FFA2 FD82 0000 8002 0000  1 ......B......
(20)DP X/10
   ($H)
   ($H) 00FDDA/ 0002 9870 1702 7D02 0216 8DF7 9087 9871  ...p..}........q
   ($H) 00FFE2/ 8AD1 1001 0002 2ED0 ABC0 0017 E822 A0D6  ..............".."
(21)S3 X+B
(22)L*
   ($H) TRUE BREAKPOINTS
   ($H)   1  LOC=00FDDA INST=F877
   ($H)   2  LOC=00FDE4 INST=A081
   ($H)   3  LOC=00FD35 INST=2ED0
```

Figure 18-2. Debugging Session of TEST

```
(23) GO
     ($H) *BP 3 $SL=001C $P=00FDE5
(24) AR
     ($H) $R1=0001 $R2=0031 $R3=0000 $R4=0000 $R5=0000 $R6=0000
     ($H) $R7=0002 $B1=00FFAF $B2=00FFA6 $B3=000000 $B4=00FFA2
     ($H) $B5=001194 $B6=00FDDA $B7=00FFA8 $P=00FDE5 $I=0004 $S=401C
(25) S4 X+F
(26) GO
     ($H) *BP 4 $SL=001C $P=00FDE9
(27) AR
     ($H) $R1=0001 $R2=0001 $R3=0000 $R4=0000 $R5=0000 $R6=080B
     ($H) $R7=0002 $B1=00FFAF $B2=00FDFE $B3=000000 $B4=00FFA2
     ($H) $B5=001194 $B6=00FDDA $B7=00FFA8 $P=00FDE9 $I=0024 $S=401C
(28) DP $B2
     ($H)
     ($H)  00FDFE/ 0604 080B 0614 091B 4140 4553 5341 4745  ........AMESSAGE
(29) S5 X+14
(30) GO
     ($H) *B 5 $SL=001C  $P=00FDEE
(31) AR
     ($H) $R1=0001 $R2=000B $R3=0000 $R4=0000 $R5=0000 $R6=080B
     ($H) $R7=0002 $B1=00FFAF $B2=00FDFE $B3=000000 $B4=00FE02
     ($H) $B5=001194 $B6=00FDDA $B7=00FFA8 $P=00FDEE $I=0024 $S=401C
(32) DP $B4
     ($H)
     ($H) 00FFE02/ 414D 4553 5341 4745 205A 4552 4F20 4154  AMESSAGE ZERO AT
(33) DP $B4/10
     ($H)
     ($H) 00FE02/ 414D 4553 5341 4745 205A 4552 4F20 4154  AMESSAGE ZERO AT
     ($H) 00FE0A/ 4553 5420 4D45 5353 4147 4520 4F4E 4520  EST MESSAGE ONE
(34) S6 X+17
(35) L*
     ($H)TRUE BREAKPOINTS
     ($H)  1  LOC=00FDDA INST=F877
     ($H)  2  LOC=00FDE4 INST=A081
     ($H)  3  LOC 00FDE5 INST=2ED0
     ($H)  4  LOC 00FDE9 INST=A0D6
     ($H)  5  LOC=00FDEE INST=00A4
     ($H)  6  LOC=00FDF1 INST=0001
(36) GO
     ($H) *BP 6 $SL=001C $P=00FDF1
(37) AR
     ($H) $R1=0001 $R2=000B $R3=0000 $R4=0000 $R5=0000 $R6=0008
     ($H) $R7=0000 $B1=00FFAF $B2=00FDFE $B3=000000 $B4=005353
     ($H) $B5=001194 $B6=00FDDA $B7=00FFA8 $P=00FDF1 $I=0004 $S=4010
(38) DP $B4
     ($H)
     ($H) 005353/ 83C8 FF68 190E A870 0008 F851 E870 0000  ...h...p...Q.p..
(39) DH $B2+B
     ($H) 00FE09/ 4154
(40) DP FE09
     ($H)
     ($H) 00FE09/ 4154 4553 5420 4D45 5353 4147 4520 4F4E  ATEST MESSAGE ON
(41) AS B4 FE09
(42) DP $B4
     ($H)
     ($H) 00FE09/ 4154 4553 5420 4D45 5353 4147 4520 4F4E  ATEST MESSAGE ON
(43) DP $B4/10
     ($H)
     ($H) 00FE09/ 4154 4553 5420 4D45 5353 4147 4520 4F4E  ATEST MESSAGE ON
     ($H) 00FE11/ 4520 414D 4553 5341 4745 2054 574F 2020  E AMESSAGE TWO
```

Figure 18-2 (cont). Debugging Session of TEST

```
(44) GO
     ($H) ILL INST "GO"
(45) C6
(46) GO
     ($H)TEST ME
     ($H)RDY:
(47) TEST 1
     ($H) *BU 1 $SL=001B $E=00FDDA + 0000 DATA=000000
(48) DP $E/20
     ($H)
     ($H) 00FDDA/ F877 9870 1702 7D02 0216 8DF7 9087 9871  .w.p..}........q
     ($H) 00FDE2/ 8AD1 1001 A081 2ED0 ABC0 0017 E822 A0D6  ..............".."
     ($H) 00FDEA/ A570 00FF CBC0 0015 CCA4 6048 7000 0001  .p........'H ...
     ($H) 00FDF2/ 0801 1907 B870 0080 F851 6C00 0001 0F00  .....p...Q .....
(49) L*
     ($H)TRUE BREAKPOINTS
     ($H)  1  LOC=00FDDA  INST=F877
     ($H)  2  LOC=00FDE4  INST=A081
     ($H)  3  LOC=00FD35  INST=2ED0
     ($H)  4  LOC=00FDE9  INST=A0D6
     ($H)  5  LOC=00FDEE  INST=CCA4
(50) C*
(51) L*
     ($H) INACTIVE BP "L*"
(52) DP $E/20
     ($H)
     ($H) 00FDDA/ F877 9870 1702 7D02 0216 8DF7 9087 9871  .w.p..}........q
     ($H) 00FDE2/ 8AD1 1001 A081 2ED0 ABC0 0017 E822 A0D6  ..............".."
     ($H) 00FDEA/ A570 00FF CBC0 0015 CCA4 6048 7000 0001  .p........'H ...
     ($H) 00FDF2/ 0801 1907 B870 0080 F851 6C00 0001 0F00  .....p...Q .....
(53) S10 $E
(54) GO
     ($H) *BP 10 $SL=001C $P=00FDDA
(55) DP $P
     ($H)
     ($H) 00FDDA/ 0002 9870 1702 7D02 0216 8DF7 9087 9871  ...p ..}........q
(56) AS X $P
(57) VH X
     ($H) X=00FDDA
(58) CH X+14  CBA2
(59) DP X+14
     ($H)
     ($H) 00FDEF/ CBA2 6048 7000 0001 0801 1907 B870 0080  ..'H ..........p..
(60) DH X+25
     ($H) 00FDFF/ 080B
(61) CH X+25  100B
(62) DH X+25
     ($H) 00FDFF/ 100B
(63) S11 X+16
(64) GO
     ($H) *BP 11  $SL=001C $P=00FDF0
(65) AR
     ($H) $R1=0004 $R2=000B $R3=0000 $R4=0000 $R5=0000 $R6=0010
     ($H) $R7=0002 $B1=00FFAF $B2=00FDFE $B3=000000 $B4=00FE09
     ($H) $B5=001194 $B6=00FDDA $B7=00FFA8 $P=00FDF0 $I=000F $S=4010
(66) DP $B4/10
     ($H)
     ($H) 00FE09/ 4154 4553 5420 4D45 5353 4147 4520 4F4E  ATEST MESSAGE ON
     ($H) 00FE11/ 4520 414D 4553 5341 4745 2054 574F 2020  E AMESSAGE TWO
(67) GO
     ($H)TEST MESSAGE ON
     ($H)RDY
```

Figure 18-2 (cont). Debugging Session of TEST

At the start of the listing shown in Figure 18-2, TEST is invoked with the argument 1. TEST should write to user-out TEST MESSAGE ONE, but fails to do so. A debugging session follows. Each Debugger directive beside which a number appears is explained below by a correspondingly-numbered comment.

1. Invoke the Multi-User Debugger.

2. Set bound unit breakpoint 1 on bound unit TEST.

3. List all bound unit breakpoints.

4. Put the debugger to sleep (SP) to allow input to the group through ECL commands.

   Note that the group is back in "RDY" state, waiting for input.

5. Type the bound unit name with an argument, causing a bound unit breakpoint message to appear.

   You are now back in debug mode, ready to type in debug directives.

6. Assign temporary symbol X to the base of the program in memory.

   If the start address of the program is not offset zero, the offset value must be subracted from $E to determine the base.

   Example:

   Assume the following bound unit breakpoint message:

   ```
   *BU 2  $SL=001B  $E=00ABCD + 0023  DATA=000000
   ```

   To set X to the base of the bound unit, you would type

   ```
   AS X $E-23
   ```

   or

   ```
   AS X ABCD
   ```

7. Verify the value that has been assigned to X.

8. Display memory starting at the location assigned to X for 10 (hexadecimal) locations.

9. Set true breakpoint 1 at location X (the base of the bound unit).

10. List all true breakpoints currently set.

11. Display memory starting at location X.

    Note that the instruction F877 has been replaced by 0002--a break instruction. The original instruction has been saved in a table in Debugger workspace.

12. Reactivate the broken task by typing GO. The GO directive causes the true breakpoint to be encountered and the message to appear.

    GO must be typed from a true breakpoint; SP is not accepted at this time.

13. Display all registers. You must be at a true breakpoint for register values to be meaningful. Register values displayed at a bound unit breakpoint are not meaningful for the bound unit being debugged.

14. Display memory pointed to by $B7 for 10 (hexadecimal) locations.

15. Set true breakpoint 2 at location X + A.

16. Type GO, causing true breakpoint 2 to be encountered and the message to appear.

17. Display all registers.

18. By means of the AS directive, change the value of $R1 from 4 to 1.

    The logic of the program calls for a division by 2 to convert the number of bytes to words. Instead, the instruction at offset 9 multiplies by 2. The value of $R1 is changed to correct this mistake.

19. Display memory pointed to by $B1.

20. Display memory starting at X, to show where breakpoints are currently set.

21. Set true breakpoint 3.

22. List all currently active breakpoints.

23. GO from breakpoint 2.

24. Display all registers.

25. Set true breakpoint 4.

26. GO from breakpoint 3.

27. Display all registers.

28. Display memory pointed to by $B2.

29. Set true breakpoint 5.

30. GO from breakpoint 4.

31. Display all registers.

32. Display memory pointed to by $B4.

33. Display more of memory pointed to by $B4 than was requested for display by the previous directive.

34. Set true breakpoint 6.

35. List all currently active true breakpoints.

36. GO from breakpoint 5.

37. Display all registers.

38. Display memory pointed to by $B4.

39. Display in hexadecimal only (not in ASCII) memory at the location pointed to by $B2 + B.

40. Display memory at location FE09.

41. Assign FE09 to $B4, which was not pointing to the proper location.

42. Display memory pointed to by $B4 to confirm that the value just assigned to $B4 is correct.

43. Display more of memory pointed to by $B4.

44. GO from breakpoint 6.

    The message ILL INST "GO" means that the breakpoint must be cleared before the GO directive can be issued. The description of the Set True Breakpoint directive (earlier in this section) explains when a breakpoint must be cleared before GO can be issued.

45. Clear breakpoint 6, replacing the 0002 break instruction with the original instruction, which has been stored in Debugger workspace.

46. GO from breakpoint 6.

    There are no more break points, and the bound unit completes execution. The group is back in ECL mode, awaiting input (RDY).

47. Type in the bound unit name with argument in order to step through the program again.

    Note that the bound unit breakpoint is still set for TEST.

48. Display memory starting at the base address of the bound unit ($E).

49. List all currently active breakpoints.

    Even though the Debugger thinks that true breakpoints 1 through 5 are active, there are no 0002 instructions in memory at the specified locations. When the bound unit TEST was reinvoked, a new copy of the bound unit was loaded in memory, overwriting the version containing the breakpoint instructions. It is important to remember that true breakpoints must be reset after each invocation of a program.

50. Clear all currently active breakpoints.

51. List all currently active breakpoints.

52. Display memory starting at the base address ($E) of the bound unit.

    You can enter the expression $E only when at a bound unit breakpoint. At other times, refer to the value of $E by assigning that value a temporary symbol in the range G through Z.

53. Set true breakpoint 10.

    It is possible to reuse breakpoint numbers 1 through 6. The number 10 was chosen simply to show that higher numbers are available.

54. GO from bound unit breakpoint 1.

55. Display memory pointed to by $P (program counter).

56. Assign the value of $P to the temporary symbol X.

57. Verify the value assigned to X.

58. Change the instruction at offset 14 from LDB to LAB.

59. Display memory starting a location X + 14 of the bound unit. (The displayed change has, of course, occurred only in memory.)

60. Display the contents of memory at offset 25 of the bound unit.

61. Change the value of the memory location X + 25.

62. Display the location again to view the new contents.

63. Set true breakpoint 11.

64. GO from breakpoint 10.

65. Display all registers.

66. Display memory pointed to by $B4.

67. GO from breakpoint 11.

    Because no more breakpoints have been set, the bound unit completes execution; the group is back at the RDY state, awaiting input.

Sample Session 2

The bound unit TSTNOW, listed in Figure 18-3, is debugged with quick breakpoints. The debugging session is shown in Figure 18-4.

```
      TITLE TSTNOW          08/13/82  1334.6 edt Fri       NRS ASSEMBLER 9.00  -LAF  PAGE  0002

000046                                          *           $USOUT      !SLEW_3,=20,=1
000047                                          *           LOB         $b4,!SLEW_3
C00048  0028  C8C0 0036                                     lab         $b4,slew_3
000049  002A  E870 0014                                     ldr         $r6,=20
000050  002C  F870 0001                                     ldr         $r7,=1
000051  002E  0001                                          mcl
C00052  002F  0801                                          dc          x'0801'
000053  0030  1903                                          bez         $r1,>bb
000054                                          *
000055  0031  A851                                aa        ldr         $r2,=$r1
C00056  0032  0F82                                          b           >cc
000057  0033  2C00                                bb        ldv         $R2,0
000058                                          *
000059                                          *$c         $TRMRQ
C00060        0034                              cc        equ         $
000061  0034  0001                                          mcl
000062  0035  0103                                          dc          x'0103'
000063                                          *
000064  0036  2041                                slew      dc          z'2041'
000065  0037  584E 4F57 5020                      buffer    text        a'[NOW] '
C00066  003A  2020                                          resv        11,z'2020'
000067  0045  2020                              extra     resv        5,z'2020'
000068                                          *
C00069  004A  2041                                slew_2    dc          z'2041'
C00070  004B  5855 5345 5250                      buff_2    text        a'[USER]'
000071  004E  2020                                          resv        12,z'2020'
000072  005A  2020                              extr_2    resv        5,z'2020'
C00073  005F  2041                                slew_3    dc          z'2041'
C00074  0060  5B6E 6F74 205B                      buff_3    text        a'[not [equal 2 2]] '
        0063  6571 7561 6C20
              3220 325D 5D20
C00075  0069  2020                                          resv        12,z'2020'
C00076                                          *
C00077  007S  0000                                          end         tstnow,begin
0000  ERR COUNT
```

Figure 18-3. Bound Unit TSTNOW

```
     TITLE TSTNOW          08/13/82  1334.4 edt Fri        HRS ASSEMBLER 9.00  -LAF  PAGE  0001
000001                                          title tstnow
000002                                 *
000003                                 *        COPYRIGHT, (C), 1981, HONEYWELL INFORMATION SYSTEMS.
000004                                 *
000005   0000   CBC0 0036             begin    lab      $b4,buffer
000006   0002   E870 401C                      ldr      $r6,=Z'401C'
000007   0004   1C05                          ldv      $r1,5
000008                                 *
000009   0005   0001                          mcl
000010   0006   0C08                          dc       x'0C08'
000011                                 *
000012   0007   19AA                          bnez     $r1,>aa
000013                                 *
000014                                 *        $USOUT   !SLEW,=29,=1
000015                                 *        $LDB     $B4,!SLEW
000016   0008   CBC0 002D                      lab      $b4,slew
000017   000A   E870 0010                      ldr      $r6,=29
000018   000C   F870 0001                      ldr      $r7,=1
000019   000E   0001                          mcl
000020   000F   0801                          dc       x'0801'
000021                                 *
000022          0010                   user     equ      $
000023   0010   CBC0 003A                      lab      $b4,buff_2
000024   0012   E870 401E                      ldr      $r6,=Z'401E'
000025   0014   1C05                          ldv      $r1,5
000026   0015   0001                          mcl
000027   0016   0C08                          dc       x'0C08'
000028   0017   199A                          bnez     $r1,>aa
000029                                 *        $USOUT   !SLEW_2,=31,=1
000030                                 *        LDB      $B4,!SLEW_2
000031   0018   CBC0 0031                      lab      $b4,slew_2
000032   001A   E870 001F                      ldr      $r6,=31
000033   001C   F870 0001                      ldr      $r7,=1
000034   001E   0001                          mcl
000035   001F   0801                          dc       x'0801'
000036                                 *
000037          0020                   nest     equ      $
000038   0020   CBC0 003F                      lab      $b4,buff_3
000039   0022   E870 4011                      ldr      $r6,=Z'4011'
000040   0024   1C05                          ldv      $r1,5
000041   0025   0001                          mcl
000042   0026   0C08                          dc       x'0C08'
000043                                 *
000044   0027   198A                          bnez     $r1,>aa
000045                                 *
```

Figure 18-3 (cont). Bound Unit TSTNOW

```
(S)GL086 M0040D-L3.0-07/02/0741
(S)  "CLMIN...OR *"
(S) C?
QUIT
(S)              ** M4/3.0    V10.1 **
(S)THIS SYSTEM IS BUILT ON THE NEW DIRECTORY STRUCTURE
(S)DATE TIME (E.G., 81 JAN 10 1405):  82 AUG 13 1538
(S)FRI AUG 13, 1982    15:38:08
(F)DAEMON GROUP READY!
(H)GROUP READY!
(H)$H
1  EC !CONSOLE
2  KDN
(S)RDY:
3   C :$H:
4  CWD !TERRY>LAF
(H)RDY:
5  CR TSTNOW.OK -REL -CISZ 4096 -LRSZ 512 -SZ 4
(H)RDY:
6   C :$S:
7  ATERR:>LAF>DEBUG
(S)DEBUG-R300-08/13/1514
8  HB
9  PB
(S) PB= 015C03
10 SF !TERR>LAF>TSTNOW.OK
11 LB! TSTNOW
12 LB*
(S) BU! TSTNOW
13 SP
(S)RDY:


14 !TERR>LAF>TSTNOW
(S) *BU 1 $SL=001C $E=022263 +  0000   DATA=000000
15 AS X $E
16 DP X
(S)
(S) 022263/ CBC0 0036 E870 401C 1C05 0001 0C08 19AA  ...6.P@.........
17 SQ1 X+7 (AR)
18 SQ2 X+17 (AR:DH X+36/20:GO)
19 DQ3 X+27 (VH $R1:DH X+4A/20)
```

Figure 18-4. Debugging Session of TSTNOW

```
20  LQ*
    (#S)QUICK BREAKPOINTS
    (#S)   1  LOC=02226A INST=19AA (AR;GO)
    (#S)   2  LOC=02227A INST=199A (AR;DH X+36/2O;GO;GO)
    (#S)   3  LOC=02228A INST=198A (VH $R1;DH X+4A/2O;GO)
21  GO
    (#S)FRI AUG 13, 1982     15:44:15
    (#S)OPERATOR.SYSTEM.OPR
    (#S)EC: 17 0805 (IC)
    (#S)UNBALANCED QUOTATION MARKS, BRACKETS, OR PARENTHESES EXIST.  THE NEXT FUNCTION
    (#S)IS PERFORMED.  CORRECT THE DELIMITER AND RETRY.
    (#S)RDY:
22  DEBUG
    (#S)DEBUG - RDY
23  LQ*
    (#S)QUICK BREAKPOINTS
    (#S)   1  LOC=02226A INST=19AA (AR;GO)
    (#S)   2  LOC=02227A INST=199A (AR;DH X+36/2O;GO;GO)
    (#S)   3  LOC=02228A INST=198A (VH $R1;DH X+4A/2O;GO)
24  LQ*
25  LQ*
    (#S)QUICK BREAKPOINTS
    (#S)   1                          (AR;GO)
    (#S)   2                          (AR;DH X+36/2O;GO;GO)
    (#S)   3                          (VH $R1;DH X+4A/2O;GO)
26  RF QK
27   $H FO 'LPTOO
    (#H)RDY:
28   $H FR_QK TSTNOW.QK
    (#H)RDY:
29  F= $H FO
    (#H)RDY:
30  FO 'LPTOO
31  DP 15003/500
32  FQ
33  FQ
    (#S) FQ= 015003
34  FQ
35  FQ
    (#S) NO QUICK MEMORY EXISTS "FQ"
36  QT
    (#S)RDY:
```

Figure 18-4 (cont). Debugging Session of TSTNOW

Each numbered Debugger directive in Figure 18-4 is explained below by a correspondingly-numbered comment.

1. Establish standard I/O files for the system ($S) group.

2. Turn on the ready prompt. (Use of the ready prompt is optional. In this example, RDY helps to distinguish user input from system response.)

3. Change the default group id to $H.

4. Change the working directory of the $H group.

5. Create a quick disk file, specifying for control interval and logical record size the default values of Get Quick Memory (MQ) arguments.

6. Change the default group id to $S.

   To use quick break points, you must invoke the Debugger from the $S group.

7. Invoke the Multi-User Debugger. (The bound unit usually resides in SYSLIB2 and can be invoked by the simple name DEBUG.)

8. Request quick memory, using the default values.

9. Print the memory location at which quick memory begins.

10. Open the quick disk file.

11. Set bound unit breakpoint one on the bound unit TSTNOW.

12. List all bound unit breakpoints currently set.

13. Put the Debugger to sleep. This directive returns the group ($S) to the ready state, allowing you to enter ECL commands.

14. Invoke the bound unit TSTNOW. This command causes bound unit breakpoint 1 to be encountered and its breakpoint message to be displayed. The occurance of breakpoint one reactivates the debugger, which will handle all input to the $S group until GO or QT is entered.

15. Assign the value of $E to the temporary symbol X.

    Since the value of $E is the base location of the bound unit, all subsequent references to a location in the bound unit can take the form: X + offset.

16. Dump one line of memory, starting at the location associated with the temporary symbol X.

17. Set quick breakpoint 1 and its associated directive line at offset 7 in the bound unit.

18. Set quick breakpoint 2 and its associated directive line at offset 17 in the bound unit.

19. Set quick breakpoint 3 and its associated directive line at offset 27 in the bound unit.

20. List all currently active quick breakpoints and their associated directive lines.

    Note that the directive line for quick breakpoint 2 ends with two GOs. The Debugger appends GO to the end of a quick breakpoint directive line, whether or not the user has already done so. The repetition of GO causes no problems. Once a GO is encountered in a directive line, the rest of the line (whatever it may be) is ignored.

21. Go from the bound unit breakpoint. The bound unit completes execution without any visible evidence that the quick breakpoints were encountered.

22. Reinvoke the Debugger.

23. List all quick breakpoints currently set and their associated directive lines.

24. Clear the quick breakpoints just listed.

25. List all quick breakpoints currently set and their associated directive lines.

    Note that although quick breakpoints 1, 2, and 3 are no longer set, their directive lines remain for future use. The clear breakpoint directive does not clear directive lines.

26. Close the quick disk file currently in use.

27. From the $H group, change user-out to the line printer.

28. From the $H group, invoke the Debugger utility PR_QK to print the information written to the quick disk file TSTNOW.QK. The print-out of this information is shown in Figure 18-5.

29. From the $H group, change user-out back to its original device.

30. Change the user-out of the (default) $S group to the line printer. In this case, FO !LPT00 is a Debugger directive.

31. Dump 500 words of memory, starting at the location displayed earlier by the PQ directive as the start of quick memory (see step 9). A print-out of this dump is shown in Figure 18-6.

32. Change user-out of the $S group back to its original device.

33. Print the start of quick memory.

34. Return the quick memory block.

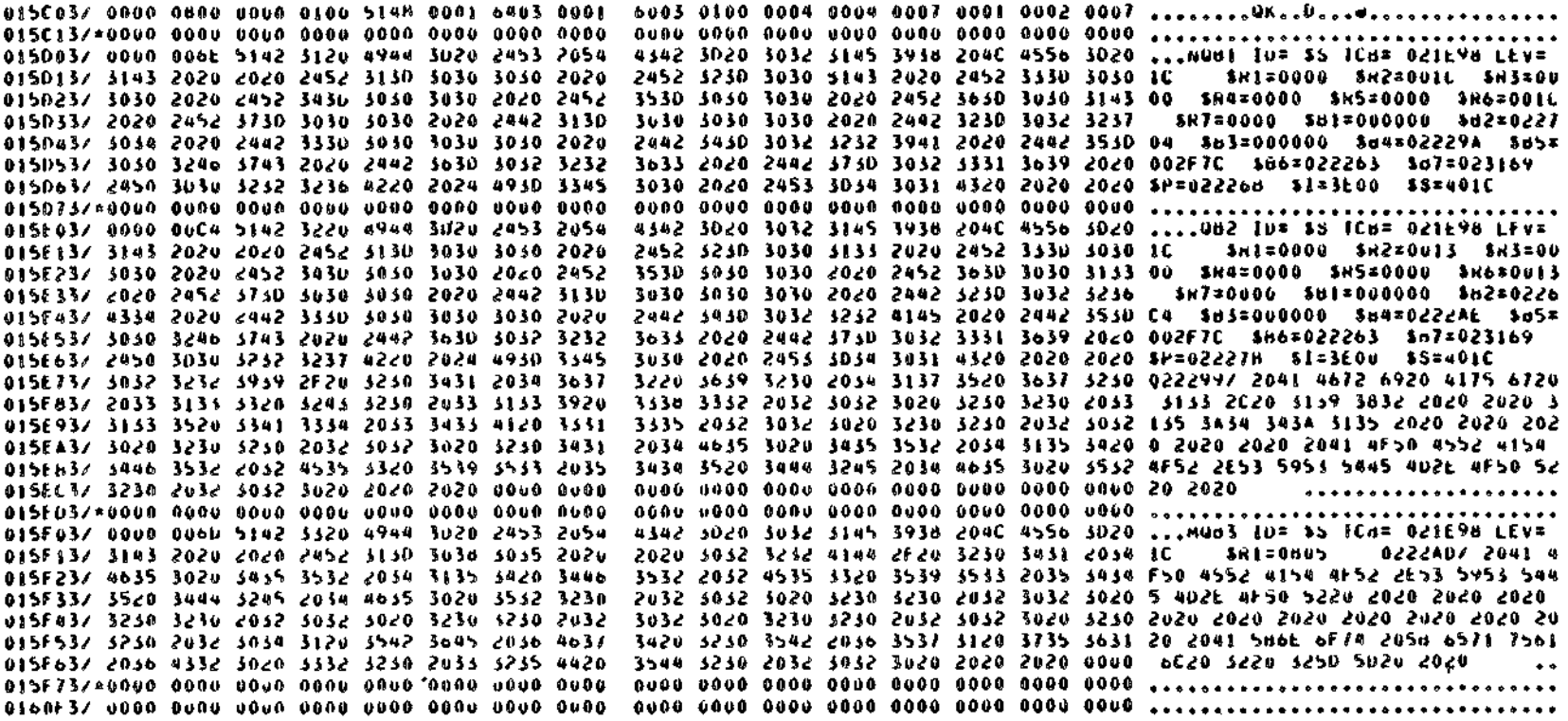
35. Print the start of quick memory. A message is returned verifying that the quick memory has been returned by the RQ directive.

36. Abort the Debugger from the $S group.

```
^TERRY>LAP>TSTNOW.QK
***DUMP OF DATA GENERATED BY MOD 400 MULTI-USER DEBUG QUICK BREAKPOINTS***

DATA FROM MEMORY BUFFER     1
 UB1 ID= $S TCB= 021E9A LEV= 1C    $R1=0000  $R2=001C  $R3=0000  $R4=0000  $R5=0000  $R6=001C  $R7=0000  $B1=000000  $B2=022704  $B3=000
00  $B4=02229A  $R5=002F7C  $B6=022263  $B7=023169  $P=02C268  $I=3E00  $S=401C
 UB2 ID= $S TCB= 021E9A LEV= 1C    $R1=0000  $R2=0013  $R3=0000  $R4=0000  $R5=0000  $R6=0013  $R7=0000  $B1=000000  $B2=0226C4  $B3=000
00  $B4=022CAE  $R5=002F7C  $R6=022263  $B7=023169  $P=022278  $I=3E00  $S=401C     022299/ 2041 4672 6920 4175 6720 5153 2C20 5159 3a3d
2020 2020 3135 3434 343A 3135 2020 2020 2020 2020 2020 2041 4F50 4552 4154 4F52 2E53 5953 5445 402E 4F50 5220 2020
 UB3 ID= $S TCB= 021E9A LEV= 1C    $R1=0805    0222AD/ 2041 4F50 4552 4154 4F52 2E53 5953 5445 402E 4F50 5220 2020 2020 2020 2020 2020 2
20 2020 2020 2020 2020 2041 5B6E 6F74 2050 6571 7561 0C20 3220 325D 5020 2020
EOF
```

Figure 18-5. Contents of Quick Disk File TSTNOW.QK



```
015C03/ 0000 0800 0000 0100 514A 0001 6A03 0001  6003 0100 0004 0004 0007 0001 0002 0007 ........QK..`.....
015C13/*0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000 ................................
015D03/ 0000 006E 5142 5120 4944 3020 2453 2054  4342 3D20 3032 3145 3938 204C 4556 3D20 ...NUBI ID= $S ICB= 021E98 LEV=
015D13/ 3143 2020 2020 2452 3130 3030 3030 2020  2452 5230 3030 5143 2020 2452 3330 3030 1C     $R1=0000  $R2=001C  $R3=00
015D23/ 3030 2020 2452 3436 3030 3030 2020 2452  3530 3030 3030 2020 2452 3630 3030 3143 00  $R4=0000  $R5=0000  $R6=001C
015D33/ 2020 2452 3730 3030 3030 2020 2442 3130  3030 3030 3030 2020 2442 3230 3032 3237   $R7=0000  $B1=000000  $B2=0227
015D43/ 3034 2020 2442 3330 3030 3030 3030 2020  2442 3430 3032 3232 3941 2020 2442 3530 04  $B3=000000  $B4=02229A  $B5=
015D53/ 3030 3246 3743 2020 2442 3630 3032 3232  3633 2020 2442 3730 3032 3331 3639 2020 002F7C  $B6=022263  $B7=023169
015D63/ 2450 3030 3232 3236 4220 2024 4930 3345  3030 2020 2453 3034 3031 4320 2020 2020 $P=02226B  $I=3E00  $S=401C
015D73/*0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000 ................................
015E03/ 0000 00C4 5142 3220 4944 3D20 2453 2054  4342 3D20 3032 3145 3938 204C 4556 3D20 ....UB2 ID= $S ICB= 021E98 LEV=
015E13/ 3143 2020 2020 2452 3130 3030 3030 2020  2452 5230 3030 3133 2020 2452 3330 3030 1C     $R1=0000  $R2=0013  $R3=00
015E23/ 3030 2020 2452 3430 3030 3030 2020 2452  3530 3030 3030 2020 2452 3630 3030 3133 00  $R4=0000  $R5=0000  $R6=0013
015E33/ 2020 2452 3730 3030 3030 2020 2442 3130  3030 3030 3030 2020 2442 3230 3032 3236   $R7=0000  $B1=000000  $B2=0226
015E43/ 4334 2020 2442 3330 3030 3030 3030 2020  2442 3430 3032 3232 4145 2020 2442 3530 C4  $B3=000000  $B4=0222AE  $B5=
015E53/ 3030 3246 3743 2020 2442 3630 3032 3232  3633 2020 2442 3730 3032 3331 3639 2020 002F7C  $B6=022263  $B7=023169
015E63/ 2450 3030 3232 3237 4220 2024 4930 3345  3030 2020 2455 3D34 3031 4320 2020 2020 $P=02227B  $I=3E00  $S=401C
015E73/ 3032 3232 3939 2F20 3230 3431 2030 3637  3220 3639 3230 2034 3137 3520 3637 3230 0222997/ 2041 4672 6920 4175 6720
015E83/ 2033 3131 3320 3245 3230 2033 3133 3920  3330 3332 2032 3032 3020 3230 3230 2033  3133 2020 3139 3832 2020 2020 3
015E93/ 3133 3520 3341 3334 2033 3435 4120 3531  3335 2032 3032 3020 3230 3230 2032 3032 135 3A34 343A 3135 2020 2020 202
015EA3/ 3020 3230 3230 2032 3032 3020 3230 3431  2034 4635 3020 3435 3532 2034 3135 3420 0 2020 2020 2041 4F50 4552 4154
015EB3/ 3446 3532 2032 4535 3320 3539 3533 2035  3430 3520 3446 3245 2034 4635 3020 3532 4F52 2E53 5953 5845 402E 4F50 52
015EC3/ 3230 2032 3032 3020 2020 2020 0000 0000  0000 0000 0000 0006 0000 0000 0000 0000 20 2020      ..................
015ED3/*0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000 ................................
015F03/ 0000 006D 5142 3320 4944 3020 2453 2054  4342 3D20 3032 3145 3938 204C 4556 3D20 ...MUB3 ID= $S ICB= 021E98 LEV=
015F13/ 3143 2020 2020 2452 3130 3038 3035 2020  2020 3032 3242 4144 2F20 3230 3431 2034 1C     $R1=0805    0222AD/ 2041 4
015F23/ 4635 3020 3435 3532 2034 3135 3420 3446  3532 2032 4535 3320 3539 3533 2035 3430 F50 4552 4154 4F52 2E53 5953 544
015F33/ 3520 3444 3245 2034 4635 3020 3532 3230  2032 3032 3020 3230 3230 2032 3032 3020 5 402E 4F50 5220 2020 2020 2020
015F43/ 3230 3230 2032 3032 3020 3230 3230 2032  3032 3020 3230 3230 2032 3032 3020 3230 2020 2020 2020 2020 2020 2020 20
015F53/ 3250 2032 3034 3120 3542 3645 2036 4637  3420 3230 3542 2036 3537 3120 3735 3631 20 2041 5A6E 6F72 205B 6571 7561
015F63/ 2036 4332 3020 3332 3230 2033 3235 4420  3544 3230 2032 3032 3020 2020 2020 0000  6C20 3220 3250 5D20 2040      ..
015F73/*0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000 ................................
0160F3/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000 ................................
```

Figure 18-6. Dump of Quick Memory

Sample Session 3

The code being debugged is in lower (system) memory, and deals with a semaphore related to all console I/O. The true breakpoint set in the following debugging session, shown in Figure 18-7, is subsequently encountered at each I/O request made to the console.

```
(S)GCOS6 MOD400-L3.0-07/02/0741
($)  "CLMIN...OR *"
(S) C?
QUIT
(S)              ** M4/3.0    V10.1 **
(S)THIS SYSTEM IS BUILT ON THE NEW DIRECTORY STRUCTURE
(S)DATE TIME (E.G., 81 JAN 10 1405):  82 AUG 13 1557
(S)FRI AUG 13, 1982     15:57:08
(F)DAEMON GROUP READY!
(H)GROUP READY!
(H):H
1  C :H:
2 CWD ^TERRY>LAF
(H)RDY:
3 LS *.OK -BF
(H)


 DIRECTORY:  ^TERRY>LAF
(H)
(H)   TSTNOW.OK      R        64
(H)   SAMPLE.OK      R        48
(H)
(H) TOTAL SECTORS         112
(H)RDY:
4  C :G:
5 CWD ^TERRY>LAF
6 EC !CONSOLE
7 RDN
(S)RDY:
8 DEBUG
(S)DEBUG-R300-08/13/1514
9 MG
10 PQ
(3) PQ= 015C03
11 SF SAMPLE.OK
12 DP 131D
(S)
(S) 00131D/ CBC0 FF52 D3C0 1994 D3C0 20C8 D3C0 1C5E  ...R..... ....^
13 SG3 131D (AR)
```

Figure 18-7. Debugging Session (Example 3)

```
14  LQ*
    ($S)3UICK BREAKPOINTS
    ($S)  3  LOC=00131D INST=CBCD (AR:GO)
15   $H LSR
    ($H)^TERRY>LAF
    ($H)^KB>SYSLIB1
    ($H)^KB>SYSLIB2
    ($H)RDY:
16   $H LWD
    ($H)^TERRY>LAF
    ($H)RDY:
17  CQ*
18  LQ*
    ($S)QUICK BREAKPOINTS
    ($S)  3                          (AR:GO)
19  RF QK
20  RQ
21  FQ
    ($S) NO QUICK MEMORY EXISTS "PQ"
22  GT
    ($S)RDY:
23   $H FQ 'LPT00
    ($H)RDY:
24   $H PR_QK SAMPLE.QK
    ($H)RDY:
```

Figure 18-7 (cont). Debugging Session (Example 3)

Each numbered directive in Figure 18-7 is explained below by a correspondingly-numbered comment.

1. Change default group id to $H.

2. Change working directory of $H.

3. List all quick disk files in the directory.

4. Change default group id to $S.

   To use quick breakpoints, you must invoke the Debugger from the $S group.

5. Change working directory of $S

6. Establish standard I/O files for the $S group.

7. Turn on the ready prompt. (Use of the ready prompt is optional. In this example, RDY helps to distinguish user input from system response.)

8. Invoke the Debugger.

9. Request quick memory, using the default values.

10. Print the start of quick memory.

11. Open the quick disk file SAMPLE.QK.

12. Dump a line of memory, starting at location 131D. This location is in system memory.

13. Set quick breakpoint 3 at location 131D, specifying a directive line.

    As already mentioned, breakpoint 3 will be encountered at each I/O request made to the console.

14. List all quick breakpoints currently set and their associated directive lines.

15. List search rules for the $H group.

16. List the current working directory for the $H group.

    Even though the Debugger in running in the $S group, a task running in $H (steps 15 and 16) can encounter a true breakpoint.

17. Clear all quick breakpoints.

    The Debugger has remained active in the $S group while quick breakpoints were encountered by a task running in the $H group (steps 15 and 16). Input to the $S group is still being handled by the Debugger.

18. List all quick breakpoints currently set and their associated directive lines.

    Note that no quick breakpoints are currently set. There is, however, a directive line ready to be used for quick breakpoint 3. The CQ directive does not clear directive lines.

19. Close the currently active quick disk file.

20. Return the quick memory requested by the MQ directive (step 8).

21. Print the start of quick memory. The message shows that quick memory has been returned.

22. Abort the Debugger from the $S group.

23. Change user-out for the $H group to the line printer.

24. From the $H group use the Debugger utility PR_QK to print information stored in the quick disk file. A print-out of this information is shown in Figure 18-8.

```
 *TERRY>LAF>SAMPLE.OK
 ***DUMP OF DATA GENERATED BY MUD 400 MULTI-USER DEBUG QUICK BREAKPOINTS***

DATA FROM MEMORY BUFFER     1
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=009E  $R3=0009  $R4=0002  $R5=0001  $R6=2453  $R7=0002  $B1=0081F6  $B2=008403  $B3=001
A5  $B4=023105  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0002  $R4=002A  $R5=0023  $R6=2453  $R7=008C  $B1=0086F7  $B2=001237  $B3=000
77  $B4=0011A6  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0003  $R4=0001  $R5=00F7  $R6=0014  $R7=4101  $B1=0083EE  $B2=001221  $B3=000
77  $B4=0011CB  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=0014  $R7=0000  $B1=0082F2  $B2=0011CH  $B3=023
F5  $B4=0233F5  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0003  $R4=0001  $R5=00F7  $R6=0022  $R7=4101  $B1=0082B3  $B2=001221  $B3=000
77  $B4=0011CB  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=0022  $R7=0000  $B1=0086B5  $B2=0011CH  $B3=023
F5  $B4=0233F5  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00WE  $R3=0009  $R4=0002  $R5=0001  $R6=0022  $R7=0002  $B1=0086CD  $B2=008403  $B3=001
A5  $B4=023105  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B

DATA FROM MEMORY BUFFER     2
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0002  $R4=0052  $R5=0023  $R6=2448  $R7=008C  $B1=00846C  $B2=001237  $B3=000
77  $B4=0011A6  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=0048  $R3=0009  $R4=0009  $R5=0001  $R6=2448  $R7=0002  $B1=00864F  $B2=0086CD  $B3=001
A5  $B4=048623  $B5=001313  $B6=0001A7  $B7=021A3C  $P=00131F  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0003  $R4=0001  $R5=00F7  $R6=000B  $R7=4102  $B1=000360  $B2=001221  $B3=000
77  $B4=0011CB  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1H     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=000B  $R7=0000  $B1=0087UE  $B2=0011CH  $B3=023
15  $B4=023215  $B5=001313  $B6=0001A7  $B7=022A3C  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0003  $R4=0001  $R5=00F7  $R6=000C  $R7=4102  $B1=000020  $B2=001221  $B3=000
77  $B4=0011CB  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1H     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=000C  $R7=0000  $B1=00892E  $B2=0011CH  $B3=023
15  $B4=023215  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
 QB3 ID= $$ TCB= 000777 LEV= 1B     $R1=0000  $R2=00FF  $R3=0003  $R4=0001  $R5=00F7  $R6=000C  $R7=4102  $B1=00868E  $B2=001221  $B3=000
77  $B4=0011CB  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=401B
```

18-95

C205-00

Figure 18-8. Dump of Quick Disk File SAMPLE.OK

```
DATA FROM MEMORY BUFFER     3
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=000C  $R7=0000  $B1=0006351  $B2=0011C8  $B3=023
15  $B4=023215  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0008  $R4=0009  $R5=01A8  $R6=2448  $R7=0001  $B1=0000444  $B2=021E83  $B3=001
A7  $B4=040620  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0008  $R2=00FE  $R3=0003  $R4=0001  $R5=00F7  $R6=0006  $R7=4101  $B1=0006AA8  $B2=001221  $B3=000
77  $B4=0011C8  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=0006  $R7=0000  $B1=0006570  $B2=0011C8  $B3=023
15  $B4=023215  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=004E  $R3=0009  $R4=0002  $R5=0001  $R6=0006  $R7=0002  $B1=0006746  $B2=0006CD  $B3=001
A5  $B4=022015  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0002  $R4=0044  $R5=0023  $R6=2448  $R7=006C  $B1=0006161  $B2=001257  $B3=000
77  $B4=0011A6  $B5=001313  $B6=0001A7  $B7=021C3C  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=0044  $R3=0009  $R4=0009  $R5=0001  $R6=2448  $R7=0002  $B1=0006496  $B2=0006CD  $B3=001
A5  $B4=040623  $B5=001313  $B6=0001A7  $B7=021A3C  $P=00131E  $I=3E04  $S=4018

DATA FROM MEMORY BUFFER     4
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0003  $R4=0001  $R5=00F7  $R6=0008  $R7=4102  $B1=0006904  $B2=001221  $B3=000
77  $B4=0011C8  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=000B  $R7=0000  $B1=0006EF  $B2=0011C8  $B3=023
15  $B4=023215  $B5=001313  $B6=0001A7  $B7=022A3C  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0008  $R4=0009  $R5=01A8  $R6=2448  $R7=0001  $B1=0006351  $B2=021E83  $B3=001
A7  $B4=040620  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0003  $R4=0001  $R5=00F7  $R6=0006  $R7=4101  $B1=0006A4F  $B2=001221  $B3=000
77  $B4=0011C8  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018

 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=0000  $R3=0002  $R4=0003  $R5=0023  $R6=0006  $R7=0000  $B1=0006385  $B2=0011C8  $B3=023
15  $B4=023215  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=004E  $R3=0009  $R4=0002  $R5=0001  $R6=0006  $R7=0002  $B1=0006104  $B2=0006CD  $B3=001
A5  $B4=022015  $B5=001313  $B6=0001A7  $B7=0007F8  $P=00131E  $I=3E04  $S=4018
 QB3 ID= $8 TCB= 000777 LEV= 1B     $R1=0000  $R2=00FE  $R3=0002  $R4=002A  $R5=0023  $R6=2453  $R7=006C  $B1=0003EE  $B2=001237  $B3=000
77  $B4=0011A6  $B5=001313  $B6=0001A7  $B7=021C3C  $P=00131E  $I=3E04  $S=4018
EOF
```

Figure 18-8 (cont). Dump of Quick Disk File SAMPLE.QK

Section 19
REQUESTING AND USING MEMORY DUMPS

This section provides procedures for requesting memory dumps, as well as procedures for analyzing, interpreting, and resolving errors using memory dumps. The following memory dump utilities are described:

- MDUMP
- DPEDIT
- DCP

MDUMP UTILITY

The MDUMP is a stand alone utility that does not run under the Executive. MDUMP may be used when it is not possible or practical to use the debug utility dump facility.

MDUMP Requirements

To use MDUMP, you need a disk that contains an MDUMP bootstrap record on sector 0, and a file (DUMPFILE) large to contain the complete memory image. The Create Volume command is used to prepare this disk (see "Preparing for MDUMP", below).

To dump memory to the disk file, bootstrap the prepared disk as described under "Procedure for Using MDUMP," below. This procedure loads and executes MDUMP. When MDUMP terminates, an image of memory is contained in DUMPFILE.

This file can be edited and printed using the Dump Edit utility, also described later in this section.

Preparing to Execute MDUMP

Before loading the program for which a memory dump is required, enter the Create Volume command:

```
CV path { -MDUMP nnnn } { -BOOT  X'hhhh' }
        { -MD    nnnn } { -BT  X'hhhh'   }
```

ARGUMENTS:

path

Designates the pathname to the disk volume being prepared for MDUMP. The pathname may be !sympd or !sympd>volid. If >volid is specified, the volume label is checked. The volume must have been previously formatted via a Create Volume command. (This command is described in detail in the Commands manual.) The volume can contain other data.

```
{ -MDUMP nnnn }
{ -MD nnnn    }
```

Writes the MDUMP bootstrap record to the volume specified in the path argument and allocates a file (DUMPFILE) large enough to contain nnnn 4K word modules to be dumped. The resulting dump volume may be used for any configuration of memory less than or equal to the value nnnn x 4K words.

```
{ -BOOT  X'hhhh' }
{ -BT  X'hhhh'   }
```

Creates bootstrap records and intermediate loader records and writes them to disk sectors 0 through 6. The optional X'hhhh' field defines certain available bootstrap options. See the Commands manual for details.

NOTE

This argument can be used in conjunction with the -MDUMP argument to obtain a combination bootstrap/MDUMP (described below).

Procedure for Using MDUMP

Once an executing program encounters a problem or a halt occurs, you can obtain a memory dump by taking the following actions:

1. Bootstrap MDUMP, which then sends the memory dump to the disk file DUMPFILE.

2. Rebootstrap the system.

3. Use the Dump Edit utility program (DPEDIT) to print all or a portion of the memory dump from the disk volume that contains MDUMP's output.

Procedure for Bootstrapping MDUMP

To bootstrap the MDUMP bootstrap record into memory, perform the procedure shown below. MDUMP then transfers to the disk file (DUMPFILE) the amount of memory image specified in the -MDUMP argument of the Create Volume command.

1. Mount the disk containing the MDUMP bootstrap routine on the device to be used in bootstrapping.

2. Press Stop and CLear.

3. Set the P-register to 0004_{16}.

4. Enter the channel number of the bootstrap device (i.e., the disk mounted in step 1) in register R1.

   If -BT was specified when creating the MDUMP dump device, bit 12 must be on in the R1 value (i.e., R1 is set to CCC8, where CCC is the channel number). This causes the MDUMP bootstrap record to be selected.

5. Enter the initial address of the memory area into which MDUMP is to be held in register B1. MDUMP requires as much memory as will hold one sector of the disk device type on which it is stored. The initial address of B1 should be at least 100_{16} to ensure that hardware dedicated locations are not overlayed.

6. Press Load, then Execute. MDUMP is read into the memory location specified in step 5 above, and dumps the amount of memory image that fills DUMPFILE. The dump is complete when an end-of-job halt occurs (see Table 19-1).

NOTE

The size of DUMPFILE is limited by the capacity of the storage device. A maximum of 120K of memory can be stored on a diskette file.

MDUMP Halts

No messages are issued during execution of MDUMP. If a halt occurs during execution, the contents of the P-register and R6 register must be displayed to determine the significance of the halt, as indicated in Table 19-1.

Table 19-1. MDUMP Halts

Register Contents			
P-Register	R6 Register	Condition	Operator Action
003E	=0	End of job	No operator action required. For information only.
003E	^=0	Disk error	Reboot MDUMP. (R6 contains the disk status word.)
03nn	=0	Trap handler error has occurred.	For a description traps, identified by nn, see Appendix A and System Messages.

Address relative to the initial address of MDUMP as stored in memory.

DUMP EDIT UTILITY (DPEDIT)

Dumps produced by the Dump Edit utility are written to the user out file, which must be capable of receiving a 132-character line.

There are two sources of dumps:

- Files created by the previous execution of the MDUMP utility. (All or selected portions of the file can be dumped.)
- Main memory. (A dump of main memory allows you to determine the configuration under which Dump Edit is executing.)

Dumps produced by Dump Edit may be logical (edited format) dumps or physical (memory image format) dumps. Control arguments in the DPEDIT command (described later in this section) allow you to request either a logical or physical dump. If these control arguments are omitted, execution of Dump Edit produces a full logical dump followed by a full physical dump.

Logical and physical dumps are printed in both hexadecimal and ASCII notation. Duplicate lines, if any, are suppressed. Suppressed lines are designated as described under "Dump Edit Line Format".

Page Header

The page heading contains the following information:

- Indicates whether the dump is from main memory or a dump file
- The date and time of the edit
- The version of DPEDIT used
- The version of the system DPEDIT is executing on
- The pool and group currently being dumped for a logical dump
- The page number.

Dump Edit Line Format

The format of a basic dump edit line for both logical and physical dumps is as follows:

Columns	Content
1-6	Six hexadecimal digits designating the starting physical (real) address of the line of dump information. The hexadecimal digit in print position 6 is always 0. This forces the dump line to agree with the template printed at the heading of each page.
7	Slash (/)
8	Blanks
9-14	Six hexadecimal digits designating the starting virtual address of the line of dump information.
18-98	Sixteen consecutive words. Each word is represented by four hexadecimal digits and is followed by a space.
99-100	Blanks
101-132	ASCII representation of the previous group of 16 consecutive words. A byte that is not printable is designated by a period (.).
1-11	Blanks
12-93	* * * * * * * * * * * * (indicates one or more duplicate lines)
94-132	Blanks

Physical Dumps

In a physical dump, the leftmost six columns of data designate real memory addresses. When the Memory Management Unit (MMU) is in use, there may be ranges of invalid virtual addresses in a physical dump from main memory. When an invalid virtual address is encountered, a message interrupts the listing of memory locations, specifying the invalid virtual address and the physical address for which no valid virtual address exists.

The virtual address is displayed whenever possible. If it does not appear, it means that the virtual and physical addresses are the same (in low memory), or that DPEDIT could not discover the virtual address corresponding to a given physical address. The listing of real locations resumes when the valid virtual address is known. The numerical sequence of real memory addresses, before and after the message, is unbroken.

A physical dump from an external dump file does not display invalid virtual address messages, and the left column of addresses is an uninterrupted continuum of physical addresses.

A physical memory dump in Figure 19-1 was produced by Dump Edit in response to the command:

```
DPEDIT ^DMPVOL>DUMPFILE -NL -TO X'0731'
```

Logical Dump Format

By means of DPEDIT control arguments, the user can select the taste groups about which logical dump supplies information. File system information can also be selected.

The main addresses in a logical dump are virtual addresses (columns 9-14). The leftmost six columns of data are physical addresses, and will be displayed whenever they differ from the virtual addresses. This applies to dumps of disk files as well as to dumps of main memory. For disk files, Dump Edit calculates the virtual address in the same way as the Memory Management Unit would under the same conditions.

Logical Dump Content

The arrangement of information in a logical dump is described in the following paragraphs and illustrated in Figure 19-1.

SYSTEM SUMMARY

The information contained in a logical dump includes:

- Location and contents of hardware-dedicated main storage
- System time of dump

- Time of system boot
- Time of power-fail restart (if it occurred)
- Hardware configuration
- Location and contents of System Control Block (SCB)
  - Model number of central processor
  - Presence (or absence) of the Commercial Instruction Processor, the Scientific Instruction Processor, and the Memory Management Unit
  - Value of the real-time clock scan cycle
  - Presence (or absence) of an operator's terminal
  - High address of virtual memory
  - High address of physical memory
- Software Configuration
  - Name and version of operating system
  - Presence (or absence) of the error message library
  - Size of trap save area (TSA)
  - Size of interrupt save area (ISA)
  - Number of indirect request blocks (IRBs) in IRB pool
  - Presence (or absence) of the batch task group.

```
MDUMP FILE  DUMP      1982/06/28 1532:49.8       DUMPEDIT- 3.0-06/22/1713          GCOS6 MOD400-L3.0-04/02/1430          PAGE 0001

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

HARDWARE  DEDICATED LOCATIONS
        000000/  83C0 FFFF 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        000010/  0000 0100 0000 0000 0006 0005 0004 0000  0000 01AD 0000 0000 0000 0000 0000 0000  ................................
        000020/  0000 0000 0000 0001 0000 0101 0000 0103  0000 0105 0000 0107 0000 0109 0000 010B  ................................
        000030/  0000 0100 0000 010F 0000 0111 0000 0113  0000 0115 0000 0117 0000 0119 0000 011B  ................................
        000040/  0000 011D 0000 011F 0000 0121 0000 0123  0000 0125 0000 0127 0000 0129 0000 0128  ...........!...#...%...'...)...+
        000050/  0000 012D 0000 012F 0000 0131 0000 0133  0000 0135 0000 0137 0000 0139 0000 013B  ...-.../...1...3...5...7...9...;
        000060/  0000 013D 0000 0139 0000 0141 0000 0143  0000 0145 0000 0147 0000 0149 0000 014B  ...=...9...A...C...E...G...I...K
        000070/  0000 014D 0000 014F 0000 0151 0000 0153  0000 0155 0000 0157 0000 0159 0000 2475  ...M...O...Q...S...U...W...Y...u
        000080/  0000 0182 0000 0556 0000 0568 0000 06A1  0000 0503 0000 0000 0000 0882 0000 0000  .......V...h....................
        000090/  0000 0000 0000 EFA9 0000 BF3E 0000 99F0  0000 9B0E 0000 9C2C 0000 9D4A 0000 9E68  ...........>.............,...J...h
        0000A0/  0000 98D2 0000 A030 0000 A158 0000 A279  0000 A5BC 0000 A6DA 0000 A7F8 0000 A916  ........0...X...y...............
        0000B0/  0000 A0F3 0000 AF03 0000 B0A8 0000 B18A  0000 B2DE 0000 AA38 0000 0000 0000 0000  ...................8............
        0000C0/  0000 0000 0000 0000 0000 0000 0000 D6C6  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        0000D0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        000100/  0000 0100 8C53 B2D3 3E20 3302 88B4 ABC0  007A AF83 CFE2 0BC0 0034 9885 D851 0570  .....8..> 0.....z.......4...0.p


TIME OF SYSTEM DUMP     1982/05/06 1859:11.4
TIME OF SYSTEM BOOT     1982/05/06 1404:11.1


SYSTEM CONTROL BLOCK     000180
        000180/  0000 0000 2020 0001 0C08 8385 5348 5339  3231 3032 312C 434F 5059 5249 4748 542C  ....  ......8H3921021,COPYRIGHT,
        000190/  2843 2920 484F 4E45 5957 454C 4C49 4E46  4F52 4D41 5449 4F4E 5359 5354 454D 532C  (C),HONEYWELLINFORMATIONSYSTEMS,
        0001A0/  494E 432E 2C31 3933 3000 2056 392E 3020  3034 2F30 322F 3134 3330 0000 8882 0000  INC.,1980.-V9.0-04/02/1430......
        0001B0/  0912 0000 08A8 0004 D0C3 0000 0000 0000  0B90 0003 1064 0000 08A8 0005 0FE3 0140  ..................d.........@
        0001C0/  F27A 0003 B0C3 0255 ADDB FDA8 0032 0000  0000 66F0 0000 0000 0003 152C 000F FFFF  .z.....U.....2.............,....
        0001D0/  0000 0003 0005 0F85 0000 0B90 0000 8705  0000 4D42 0025 FA7F FFFC 1000 0000 0001  .h.C.................MB.%..........
        0001E0/  0000 0000 0000 0000 0CA1 0000 0000 0000  0000 2442 FFFE 0000 F4AA 2800 0000 1C4A  ..................$B......(....J
        0001F0/  0000 0486 0000 2409 000F FFFF 0004 E1E3  0000 0000 0000 0000 0000 0000 E000 0000  ......$.........................
        000200/  0000 0000 0000 0000 0000 0000 0005 0E23  0003 0000 0427 1096 0003 C45A 0000 0000  ...............#.....'.....Z....
        000210/  0005 0003 0000 252D 000A 0000 2969 0000  0000 0000 0000 0005 05A6 0000 9807 0019  ......%-....)i..................
        000220/  0003 1CA3 3000 0000 C888 0000 0001 748B  0000 0000 0801 0000 0000 0000 0000 0000  ....0.........t.................
        000230/  0000 0000 0000 0006 0000 0000 0001 339F  0001 3667 0001 3823 0080 0001 3FD0 0025  ..............3...6g..8#....?..%
        000240/  000A 0000 0001 0001 37F2 0300 2AA2 0000  2ACA 0001 3098 0000 2078 0000 0E92 0000  ........7...*...*...0...x.......
        000250/  0E70 0000 0458 0000 0000 0000 0000 2020  2020 0004 E2F0 0000 0000 0000 0000 0000  .....X..........  ..............
        000260/  0000 0001 30D6 0005 0438 0000 0000 0000  13B5 0000 27F5 0000 27FE 0000 0000 0000  ....0....8..........'...'.......
        000270/  0000 0000 0000 0001 0000 0000 0000 454E  0000 0001 7465 0000 0000 0000 55EA 0255  ..............EN....te......U..U
        000280/  ACED E92F 0000 0000 0000 0000 0450 0005  1800 0006 0000 0001 0003 1073 0003 1049  ...)........\.............s...I
        000290/  0001 0000 0000 0000 7FFF 0001 EE18 0006  2CE3 0001 9C56 0000 0000 0000 0000 0000  ................,....V..........
        0002A0/  0000 0000 FFFF 0005 6674 0004 F2C3 0004  F523 0004 F238 0402 0003 8685 0000 0000  ........t.......#...8...........


COMMERCIAL PROCESSOR:  YES
SCIENTIFIC PROCESSOR:  YES
MEMORY MANAGEMENT UNIT IN USE:  YES
TIME (MILLISEC) BETWEEN REAL TIME CLOCK INTERRUPTS: 0032
OPERATOR'S TERMINAL: YES
HIGH PHYSICAL MEMORY ADDRESS:  0FFFFF
HIGH VIRTUAL MEMORY ADDRESS:  0FFFFF
EXTERNAL DUMPFILE'S CAPACITY:  0FFFFF

OPERATING SYSTEM:  MOD400/L3.0
OPERATING SYSTEM VERSION:  -V9.0-04/02/1430
```

LOGICAL DUMP:
GENERAL SYSTEM
INFORMATION

19-8

C20S-00

Figure 19-1. Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 153249.8      DUMPEDIT- 3.0-06/22/1713          6C036 MOD400-L3.0-84/02/1630          PAGE 0002

 REAL   VIRTUAL  0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F         ASCII REPRESENTATION

ERROR MESSAGE LIBRARY: YES
SIZE OF TRAP SAVE AREA (WORDS):  0068
SIZE OF INTERRUPT SAVE AREA (WORDS): 0043
NUMBER OF UNUSED INDIRECT REQUEST BLOCKS:  0140

BATCH GROUPS: YES
VIRTUAL ADDRESS OF BEGINNING OF BACKGROUND:  000000
VIRTUAL ADDRESS OF THE END  OF BACKGROUND:  000000
CURRENTLY ROLLED_OUT:  NO
NUMBER OF COMPLETED ROLL_OUT/ROLL_IN EVENTS: 0000
MEMORY GIVEN TO FOREGROUND FROM BACKGROUND (WORDS):  0000
```

LOGICAL DUMP GENERAL SYSTEM INFORMATION (CONT).

19-9

C205-00

Figure 19-1 (cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 1532:49.8      DUMPEDIT- 5.0-06/2C/1713          GCD86 MOD400-L3.0-08/02/1430          PAGE 0003

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

MEMORY POOL DEFINITIONS                                          MEMORY POOLS STATUS
  POOL      START       END       SIZE      PHYSICAL             TOTAL_UNUSED     MAXIMUM_UNUSED     NUMBER_OF   NUMBER
  NAME      ADDRESS     ADDRESS   (WORDS)   START                  (WORDS)     CONTIGUOUS (WORDS)   FRAGMENTS  OF_USERS
   $$       051540      0517FF    02C2C0    051540                 00DAA0          00A580           000048     000002
   SM       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L0       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L1       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L2       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L3       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L4       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L5       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L6       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L7       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L8       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   L9       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   LA       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   LB       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   LC       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   LD       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   ZZ       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   CC       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   AB       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   SP       051800      0BFFFF    06E800    051800                 046900          032800           00000E     000007
   T2       060000      06FFFF    010000    0C0000                 010000          010000           000001     000000
   T1       07A000      07FFFF    010000    0D0000                 008560          0084C0           000004     000001
   BATCH    060000      09FFFF    020000    0E0000                 020000          020000           000001     000001

                                                                                                  NUMERIC INFORMATION ON ALL POOLS

MEMORY POOL ATTRIBUTES
  POOL      PROTECTED  CONTAINED  OVERLAPPED  ISOLATED  SNAPPED  SERIALLY   QUEUED MEMORY   USER'S_WRITE   PRIVILEGED
  NAME                                                          USED       MANAGEMENT      RING_NUMBER
   $$         YES        NO         NO          NO        NO       NO          NO              0            YES
   SM         YES        YES        NO          NO        YES      NO          YES             3            NO
   L0         YES        YES        NO          NO        YES      NO          YES             3            NO
   L1         YES        YES        NO          NO        YES      NO          YES             3            NO
   L2         YES        YES        NO          NO        YES      NO          YES             3            NO
   L3         YES        YES        NO          NO        YES      NO          YES             3            NO
   L4         YES        YES        NO          NO        YES      NO          YES             3            NO
   L5         YES        YES        NO          NO        YES      NO          YES             3            NO
   L6         YES        YES        NO          NO        YES      NO          YES             3            NO
   L7         YES        YES        NO          NO        YES      NO          YES             3            NO
   L8         YES        YES        NO          NO        YES      NO          YES             3            NO
   L9         YES        YES        NO          NO        YES      NO          YES             3            NO
   LA         YES        YES        NO          NO        YES      NO          YES             3            NO
   LB         YES        YES        NO          NO        YES      NO          YES             3            NO
   LC         YES        YES        NO          NO        YES      NO          YES             3            NO
   LD         YES        YES        NO          NO        YES      NO          YES             3            NO
   ZZ         YES        YES        NO          NO        YES      NO          YES             3            NO
   CC         YES        YES        NO          NO        YES      NO          YES             3            NO
   AB         YES        YES        NO          NO        YES      NO          YES             3            NO
   SP         YES        YES        NO          NO        YES      NO          YES             3            NO
   T2         YES        YES        NO          NO        NO       NO          NO              2            NO
   T1         YES        YES        NO          NO        NO       NO          NO              2            NO
   BATCH      YES        YES        NO          NO        NO       NO          NO              3            NO

                                                                                                  PROPERTIES OF ALL POOLS
```

Figure 19-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP      1982/06/28 1532:49.8      DUMPEDIT- 3.0-06/22/1713          GC036 MOD400-L3.0-04/02/1430          PAGE 0004

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

STRUCTURES FOR MEMORY POOL $5

MEMORY POOL DESCRIPTION 031049
        031040/  0000 0000 0000 0000 0000 0001 0000 0000  0000 0310 0002 0003 1045 0003 0FC3 0004  ................E.......
        031050/  AEC3 0003 0003 1180 001E 1033 0027 0FDE  186C 0000 0000 0000 0000 0000 1773 0186  ............3.'.........0..
        031060/  0944 0C3F 0048 0051 0003 198E 2424 0003  1849 0000 0000 0000 0000 0000 0000 0001  .D.?.K.Q....$$...I..............

BIT MAP  031161
        031180/  FFFF FFFC 03C0 0000 0000 3F80 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ..........?.....................
        031190/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                   *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        0311B0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0E00 0070 0000 0000 0000  ..........................p.....
        0311C0/  0000 0000 0000 0000 0000 0000 0007 FF00  0000 0000 0000 0000 0007 FC01 FFFF 8000  ................................
        0311D0/  0000 0000 0003 FF80 07FF FFFF FFFF FFFC  F800 0000 0000 0000 0000 03FF FFFF FFFF  ................................
        0311E0/  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ................................
                   *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        031200/  C7FF FFFF FFF1 F8E0 7E00 0200 0000 2070  00FF 1E00 0000 00FE 00F0 0000 0000 03FF  ..............  P...............
        031240/  FFC0 FFFF C7FC 7FF0 3FC0 FFC7 07FF 1FF1  FF00 FFC0 FC11 C000 FC00 0000 0000 0080  .........?......................
        031250/  0007 F000 1C60 0000 0000 01C0 0400 8700  E018 1E60 3878 E200 0000 0030 0007 E0E0  .....'.......0.......8x.....0....
        031260/  0080 0100 0200 C000 0000 0010 0000 0000  0800 0040 1000 0000 0000 0000 0010 0200  ...................@............
        031270/  0000 2000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0004 0000 0002 0000 0000  ..  ...............'............
        031280/  0000 0000 0000 0FFF FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ................................
```

SEGMENT DESCRIPTORS

SEGMENT	BASE	SIZE	READ	WRITE	EXECUTE
0.0	000000	002000	3	0	3
0.1	001000	002000	3	0	3
0.2	002000	002000	3	0	3
0.3	003000	002000	3	0	3
0.4	004000	002000	3	0	3
0.5	005000	002000	3	0	3
0.6	006000	002000	3	0	3
0.7	007000	002000	3	0	3
0.8	008000	002000	3	0	3
0.9	009000	002000	3	0	3
0.A	00A000	002000	3	0	3
0.B	00B000	002000	3	0	3
0.C	00C000	002000	3	0	3
0.D	00D000	002000	3	0	3
0.E	00E000	002000	3	0	3
0.F	00F000	002000	3	0	3
1.0	010000	020000	3	0	3
2.0	020000	020000	3	0	3
3.0	030000	020000	3	0	3
4.0	040000	011800	3	0	3
5.0	050000	001800	3	0	3

```
STRUCTURES FOR MEMORY POOL $8                                                    DETAILED INFORMATION FOR THE FIRST POOL $$

MEMORY POOL DESCRIPTION 031073
        031070/  0000 0000 0000 009C 0007 0003 106F 0003  0EC8 0004 C583 0008 0000 0000 0518 0BFF  ..............o.....................
```

Figure 19-1 (Cont). Memory Dump Example

```
 REAL    VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

FREE INDIRECT REQUEST BLOCK QUEUE

FREE INDIRECT REQUEST BLOCK      008BB2
        0098B0/  0000 0000 0000 9536 0000 0000 0004 F523 0003 1605 0140 0255 ADDB 8A9E 0000  .......6.........#.....a.U.....
        0098C0/  0000 0000 0000 0003 1605 0000 0000 8282 06B5 0000 EC3B 0000 04A4 0000 06A1  ....................;..........

FREE INDIRECT REQUEST BLOCK      009536
        0097530/  0000 0000 0000 EFA9 0000 0000 0000 9654 0690 0000 144F 0000 04A4 0000 05D3 0002  ..............\.....O...........
        0097540/  0000 06A1 0000 0000 0000 0000 0000 0000 05D3 0000 0000 0000 0000 0000 0000 0000  ................................

FREE INDIRECT REQUEST BLOCK      00965C
        0097650/  0000 0000 0000 0000 0000 0000 0000 000A E430 0000 0000 0000 8171 0000 0000  ..................0........q....
        0097660/  0000 0000 CA63 0004 BD10 0140 0255 ADDB 8D28 0000 0000 0000 0004 BD10 0000  .....c.....a.U...(..............
        0097670/  0000 0000 0000 0000 0000 0000 0004 FDE3 0003 87A5 0021 0000 9671 0000 0000  ..................c...!...q.....

FREE INDIRECT REQUEST BLOCK      008171
        0097170/  0000 0000 0000 0690 0000 144F 0000 04A4 0000 05D3 0002 0000 06A1 0000 0000  ..........O.....................
        0097180/  0000 0000 0000 05D3 0000 0000 0000 0000 0000 0000 0000 0004 ED43 0000 0000  ..........................C.....

FREE INDIRECT REQUEST BLOCK      008B9D
        0098980/  0438 0482 0000 0882 0000 0000 0000 0000 0000 0000 06A1 0000 0000 0000 8F0F  .8.............................
        0098A0/  0000 0000 0004 CA63 0004 BD10 0140 0255 ADDB 9110 0000 0000 0000 0000 0004 BD10  .......c.....a.U................
        0098B0/  0000 0000 0000 9536 0000 0000 0000 0004 F523 0003 1605 0140 0255 ADDB 8A9E 0000  .......6.........#.....a.U.....

FREE INDIRECT REQUEST BLOCK      00BF0F
        0098F00/  04A4 0003 0438 0482 0000 0882 0000 0000 0000 0000 0000 0000 06A1 0000 0000 0000  .....8.........................
        0098F10/  05DF 0690 0000 144F 0000 04A4 0000 05D3 0002 0000 06A1 0000 0000 0000 0000 0000  ........O.......................
        0098F20/  0000 05D3 0000 0000 0000 8D5E 9FD4 000B 1AE3 0004 DBC3 0000 8B66 100B 0003 9FE5  ..........^.............f.......

FREE INDIRECT REQUEST BLOCK      00855F
        0098500/  E043 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0004 E430 0000 0000 0000  .C.....................0......
        0098510/  7EB1 0000 0000 0000 0004 F523 0003 1605 0140 0255 ADDB 9462 0000 0000 0000 0000  ..........#.....a.U...b.........
        0098520/  0003 1605 0000 0000 0000 8D2C D6B5 0000 8354 0004 E063 0003 ADD5 0583 0000 D6C6  ..................T...c.........

FREE INDIRECT REQUEST BLOCK      007EB1
        0097ED0/  0000 0000 B6DB 0000 0000 0000 0004 CA63 0004 BD10 0140 0255 ADDB 94F8 0000 0000  ..............e.....a.U.........
        0097EE0/  0000 0000 0004 BD10 0000 0000 815C D665 0000 DC46 0000 04A4 0000 EFA9 0002  ..............\.....F...........

FREE INDIRECT REQUEST BLOCK      0086DB
        0098D00/  0000 86C6 0004 A6DD 0000 0000 0000 0005 0438 0000 0000 0000 8036 0690 0000 144F  ..............8.......6.....O
        0098E0/  0000 04A4 0000 05D3 0002 0000 06A1 0000 0000 0000 0000 0000 0000 05D3 0000 0000  ................................
        0098F0/  0000 03B3 0000 0000 0000 0004 CA63 0004 BD10 0141 0255 ADDB FE70 0000 0000 0000  ............c.....A.U...p.......

FREE INDIRECT REQUEST BLOCK      008036
        0098300/  0000 0000 0004 E430 0000 0000 0000 9410 0000 0000 0000 0004 CA63 0004 BD10 0140  .......0.................c.....@
        0098400/  0255 ADDB 9BE0 0000 0000 0000 0004 BD10 0000 0000 010C 8FCC 0C09 280B 0008  .U..........................(...

FREE INDIRECT REQUEST BLOCK      009410
        0094100/  0000 8A4D 0690 0000 144F 0000 04A4 0000 05D3 0002 0000 06A1 0000 0000 0000 0000  ...M.....O.....................
        0094200/  0000 0000 05D3 0000 0000 0000 0000 0000 0000 0000 0004 ED43 0000 0000 0000 0000  ..........................C.....

FREE INDIRECT REQUEST BLOCK      008A4D
        0098A40/  9FE5 0503 0005 1698 0000 0000 0000 0000 0000 0003 9FE5 0000 0000 0000 89F9 0000  ................................
        0098A50/  0000 0000 0004 CA63 0004 BD10 0140 0255 ADDB 9CC8 0000 0000 0000 0004 BD10 0000  .......c.....a.U................
        0098A60/  0000 0000 0000 80F3 D6B5 0000 DC46 0000 04A4 0000 EFA9 0002 0000 D6C6 0000 0000  ..............F.................
```

EXCERPT FROM FREE AND INDIRECT REQUEST BLOCK QUEUE

Figure 19-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 15:32:49.8      DUMPEDIT- 3.0-06/22/1713          GCU86 MOD400-L3.0-04/02/1630          PAGE 0022

 REAL   VIRTUAL  0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F        ASCII REPRESENTATION

SYSTEM SYMBOL TABLE
BOUND UNIT    SYMBOL      VALUE
 ZEXEC
 ZUTDUM
               ZGTMAX     0097BF  LOCATION
               ZGTMGL     0097A1  LOCATION
 ZUEXEC
               ZQESCB     00C524  LOCATION
               ZUEGLB     00C526  LOCATION
               ZUECIT     00C528  LOCATION
               ZUEMAI     00C52A  LOCATION
               ZUEMSU     00C52C  LOCATION
               ZUELSU     00C52E  LOCATION
               ZUEMLB     00C530  LOCATION
               ZUELIB     00C532  LOCATION
               ZUEFPU     00C534  LOCATION
               ZUELPL     00C536  LOCATION
               ZUEFNB     00C538  LOCATION
               ZUELML     00C53A  LOCATION
               ZUEIML     00C53A  LOCATION
               ZUEXII     00C575  LOCATION
               ZUEIMI     00C5D4  LOCATION
               ZGEMAP     00C5D2  LOCATION
               ZUEIM2     00C6D2  LOCATION
               ZUEU01     00D3D4  LOCATION
               ZUEU02     00D3D8  LOCATION
               ZUEU03     00D3E1  LOCATION
               ZUEU04     00D3E7  LOCATION
               ZGE1CP     00E0AE  LOCATION
               ZUEXEC     00E0EE  LOCATION
               ZGE11P     00E9EA  LOCATION
               ZUEUPD     00EFE3  LOCATION
               ZUEUPS     00EFE9  LOCATION
               ZUEUPX     00EFF0  LOCATION
               ZUEUPZ     00F0D3  LOCATION
               ZUEPDU     00F07E  LOCATION
               ZUEPMU     00F09A  LOCATION
               ZUEDUG     00F1D5  LOCATION
               ZUEPST     00F0AB  LOCATION
               ZUEIN3     00F0CE  LOCATION
               ZUEIN4     00F1FF  LOCATION
               ZUEGET     00F0E1  LOCATION
               ZUEPUT     00F1D0  LOCATION
               ZGETMG     00F157  LOCATION
               ZUETEG     00F1CD  LOCATION
               ZUENXT     00F1DC  LOCATION
               ZUEUMU     00F249  LOCATION
               ZUETMU     00F2C0  LOCATION
               ZGETMK     00F224  LOCATION
               ZUEVMT     00F1FC  LOCATION
               ZUPLMT     00C57E  LOCATION
               ZUELMT     00F20C  LOCATION
               ZUEMUS     00C779  LOCATION
               ZUEMAM     00F2A7  LOCATION
               ZUETCP     00F288  LOCATION
               ZUEPUC     00F2FF  LOCATION
```

EXCERPT FROM THE SYSTEM SYMBOL TABLE

Figure 19-1 (Cont). Memory Dump Example

```
 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION


RECORD LOCKING POOL CONTROL BLOCK    04E163
        04E160/  0001 0000 0000 0003 C2C3 0003 C2C5 0000  0000 001E 001E 0000 001E 001E 0000 0000  ........#...X.....
        04E170/  0000 0000 0255 ADDA 71C2 0069 0001 E1F4  0004 E1F4 0004 E1F4 0000 0000 0009 0001  .....U..q..i.....

TREE OF VOLUME, DIRECTORY, AND FILE DESCRIPTION BLOCKS

                VOLUME DESCRIPTION BLOCK
                DEPTH  00       LOCATION  000DA8
                LOGICAL RESOURCE NUMBER  0001
                ****************************
        0003C00/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 98EF 0000 0000 0000 0C04 0000 0000  ................................
        0003C10/ 0000 0000 0000 0001 0000 0000 0060 0646  0000 0000 2453 0000 0000 0001 0000 0000  ..............B....$S..........
        0003C20/ 4359 4C4F 4E20 2020 2020 2020 2020 0020  0100 0000 0004 0005 0000 0000 00A4 0000  CYLON       & . ..............
        0003C30/ FFFD 0005 0005 0005 00A0 0000 0098 0096  0000 0000 0000 0000 0013 0000 00A0 0000  ................................
        0003D40/ 0001 00A5 0000 0001 0100 0001 0000 0028  0100 0000 0028 0000 0000 0000 0000 0000  ...............(.....(..........
        0003D50/ 0000 0003 7603 0000 0000 0000 4643 4D30  3020 2020 2020 2020 0000 000E 1100 0003  ....v6......FCM00           ........
        0003D60/ 1945 0255 7CFC 8DC8 0005 1263 0000 0DA8  0000 0000 0000 0000 0000 0000 0000 0001  .%.U.....e......................

                        FILE DESCRIPTION BLOCK
                        DEPTH  01       LOCATION  000C04
                        ****************************
        0003C00/ 1945 0255 7CFC 8DC8 0005 1263 0000 0DA8  0000 0000 0000 0000 0000 0000 0000 0001  .%.U.....e......................
        0003C10/ 0000 0000 0000 0100 0000 0000 4C37 0000  0000 0001 0000 0000 5A33 4558 4543 5554  ............L7..........Z3EXECUT
        0003C20/ 4956 4553 0782 0100 0100 0000 0011 008C  0000 0000 00AE 0000 FFFD 008C 0001 0000  IVES............................
        0003C30/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0014 00AF 0000 0801  ................................
        0003C40/ 0100 0001 0000 045C 0100 0000 045D 0000  0000 0000 0000 0000 0000 0000 0000 0000  .......\.....]..................
        0003C50/ 0000 0000 2020 C000 0003 0C6D 03DA 0100  0000 0125 0000 0000 0000 0000 0000 0000  .... .....m.....%..............
        0003C60/ 0000 0000 0000 0000 0000 0C04 0000 0000  8012 0000 0001 0000 0000 0000 0000 0000  ................................

                        DIRECTORY DESCRIPTOR BLOCK
                        DEPTH  01       LOCATION  051263
                        ****************************
        051260/ 0003 0000 0000 0005 0FE3 0000 68A8 0005  1203 1800 0000 0000 0000 0000 0001 0000  ................................
        051270/ 0000 0000 0000 0000 0000 4343 0000 0000  0001 0000 0000 4D4C 2020 2020 2020 2020  ..........CC..........ML
        051280/ 2020 2020 0020 0100 0000 0AF8 0001 1088  0001 00AF 0000 FFFD 0002 0001 0000 0008  &  . ............................
        051290/ 0000 0000 0000 0000 0000 0000 0000 0005  0000 0000 0000 0019 0080 0000 0801 0100  ................................
        0512A0/ 0001 0000 0009 00A0 0000 0010 0000 0000  0000 0000 0000 0000 0005 1743 0000 0000  ..............................C....
        0512B0/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                                FILE DESCRIPTION BLOCK
                                DEPTH  02       LOCATION  051203
                                ****************************
        0511F0/ 0003 0000 0000 0005 11A3 0005 1263 0000  0000 1000 0000 0000 0000 0000 0001 0000  ..............c.................
        051200/ 0000 0000 0000 0000 0000 4343 0000 0008  0001 0000 0000 4D4C 4649 4C45 2E45 4E20  ..........CC..........MLFILE.EN
        051210/ 2020 4F63 00FC 0200 0000 0AFC 0116 0000  0000 0028 0000 FFFD 0116 0002 0000 0000    Oc.............(..*.............
        051220/ 0000 0005 0000 0000 043E 0000 0002 0000  0000 0000 0001 0009 002C 0000 0801 0100  .........>.....................
        051230/ 0002 0000 0458 0200 0000 0458 0000 0000  0005 136F 0000 0000 0005 1103 0005 112E  .....X......X.......o...........
        051240/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
```

EXCERPT FROM THE FILE SYSTEM INFORMATION

Figure 19-1 (Cont). Memory Dump Example

```
 REAL   VIRTUAL  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F        ASCII REPRESENTATION

CHAIN OF PUBLIC BUFFER POOLS
 **********************************************************


BUFFER POOL NAME: SYS256
NUMBER OF BUFFERS: 0032
BUFFER SIZE (BYTES): 0100
NUMBER OF USERS (FILES): 0002

 BUFFER POOL HEADER FOR POOL SYS256  031CA3
        031CA0/  003A 0000 0000 0003 3003 0022 5359 5332  3536 0100 0032 0032 0002 0019 0000 0000  .:.......3..."SY8256...2.2.........
        031CB0/  0000 0003 2076 0003 2226 0000 0000 0000  0000 33A7 000B 6735 0000 0000 000C F316  .... v...&.........3...g5..........
        031CC0/  0000 FF82 0000 DA4D 0000 0000 0000 12B8  0000 9990 0000 0000 0000 0000 0000 0000  ........M.......................

 BUFFER CONTROL BLOCKS FOR POOL  SYS256  031CCE
        031CC0/  0000 FF82 0000 DA4D 0000 0000 0000 12B8  0000 9990 0000 0000 0000 0000 0000 0000  ........M.......................
        031CD0/  0000 0000 0000 0000 0020 0003 23E3 0100  0000 001F 0000 0000 0001 0000 0000 0000  ......... ...#.....................
        031CE0/  0000 909E 0041 0E01 0003 25E3 0100 C75F  0000 0000 0000 0000 0003 1E7E 0003 1E7E  .....A....%.....G_.........~.....~
        031CF0/  0000 0000 0000 0000 0000 0000 0000 0000  0027 0003 2463 0100 0026 0000 0000 0000  ................'...$c...&........
        031D00/  0001 0000 0000 0000 0000 837E 0041 0E01  0003 2463 0100 CA36 0000 0000 0000 0000  ...............A....$c...6........
        031D10/  0003 1D3A 0003 1DA6 0000 0000 0000 0000  0000 0000 0000 0000 001C 0003 24E3 0100  ...:..........................$...
        031D20/  0000 001D 0000 0000 0001 0000 0000 0000  0000 86F0 0041 0E01 0003 24E3 0100 C75B  ....................A....$....G[
        031D30/  0000 0000 0000 0000 0003 20BE 0003 25B2  0000 0000 0000 0000 0000 0000 0000 0000  .......... ...#.....................
        031D40/  0028 0003 2563 0100 0000 0027 0000 0000  0001 0000 0000 0000 0000 9368 0041 0E01  .(..%c.....'...............h.A...
        031D50/  0003 2563 0100 CA37 0000 0000 0000 0000  0003 202E 0003 1CF2 0000 0000 0000 0000  ..%c...7.......... ..............
        031D60/  0000 0000 0000 0000 001E 0003 25E3 0100  0000 001D 0000 0000 0001 0000 0000 0000  ..............%.....................
        031D70/  0000 8219 0041 0E01 0003 25E3 0100 C75D  0000 0000 0000 0000 0003 1E7E 0003 20BE  .....A....%.....]..........~..... .
        031D80/  0000 0000 0000 0000 0000 0000 0000 003A  0003 2663 0100 0000 0039 0000 0000 0000  ...............:..&c.....9........
        031D90/  0001 0000 0000 0000 0000 0431 0041 0E01  0003 2663 0100 D079 0000 0000 0000 0000  ...........1.A....&c...Py..........
        031DA0/  0003 20DA 0003 1FE6 0000 0000 0000 0000  0000 0000 0000 0000 0026 0003 26E3 0100  .. ...........................&...&...
        031DB0/  0000 0025 0000 0000 0001 0000 0000 0000  0000 8282 0041 0E01 0003 26E3 0100 CA35  ...%.....................A....&....5
                                                  ***  REFER TO PHYSICAL DUMP  ***
        032300/  0003 1D1C 0003 1F0E 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

 BUFFERS FOR POOL  SYS256  0323E3
        0323E0/  00C9 0000 0000 0100 0000 0000 0020 0000  801F 00FF 647C 1CFF 64AB C600 0587 64AF  ..............  .....d|...d.....d.
        0323F0/  E48C C7FF FE8D 44FF FE6C 05D3 8010 0000  0D09 0000 0000 8765 CC01 1000 002B 0CAB  .......D..l.............e.....+..
        032400/  C600 13AF C600 05C8 C600 28AB C600 0587  64AF E4F8 7010 60FF 648C C7FF FE8D 44FF  ..........+.....d...p.`.d.....D.
        032410/  FE6C 02D3 8010 0000 0D09 0000 0000 9EB0  0000 1000 0021 0CFF 4600 21CB C600 28AC  .l.....................!...F.!...+.
        032420/  C600 0587 64AF E48C C7FF FE8D 44FF FE6C  01D3 8010 0000 0D09 0000 0000 C6DE 4D98  ....d.......D..l..............M.
        032430/  1000 000F 0CF8 4600 219B C000 0A83 8010  0000 0D09 0000 0000 C744 CA16 1000 0027  ......F.!...............D.....'
        032440/  0C00 0006 0002 0C00 5000 0600 0AFF FAFF  FD00 02F8 4600 2183 C040 4BCB C600 282C  ........P...........F.!...K...+.
        032450/  005B 8010 0000 0900 0000 02E3 1000 0015  0CAF 649F E48C C7FF FE8D 44FF FE6C 01D3  .....[..........d.......D..l..
        032460/  8010 0040 0509 0100 0000 0000 0027 1000  000D 0900 0000 0083 C5A7 9210 0000 110C  ...@.........'..................
        032470/  83C0 0001 CBC6 000A 2C00 9BB0 1000 0009  0D90 0003 3316 0000 150C AF64 9FE4 8CC7  ........+.....3.......d.......
        032480/  FFFE 8D44 FFFE 6C01 0380 1000 0009 0900  0011 9810 0000 110C 7901 FFEF CBC6 000A  ...D..l.............y.........
        032490/  2C00 9BB0 1000 0009 0D00 0003 4410 0000  150C AF64 9FE4 8CC7 FFFE 8D40 FFFE 6C01  ,.............D.......d.......D..l.
        0324A0/  0380 1000 0009 0900 0000 0083 C5A7 9210  0000 0D0C CBC6 000A 2C00 9880 1000 0009  ....................,.....,.....
        0324B0/  0D00 0003 ED10 0000 150C AF64 9FE4 8CC7  FFFE 8D40 FFFE 6C01 D380 1000 0009 0900  .............d.......D..l.........
        0324C0/  0000 00C6 DE4D 9A10 0000 210C 9FC6 0003  CBC6 000A ACC6 0003 8764 AFE4 8CC7 FFFE  .....MZ...!.............d.......
        0324D0/  8D44 FFFE 6C01 0380 1000 0009 0900 0000  0061 C267 DA10 0040 110C CBC6 000A ACC6  .D..l..............a.g...@.........
                                                  ***  REFER TO PHYSICAL DUMP  ***
        0324E0/  69ab 6564 20a0 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ined a..........................

BUFFER POOL NAME: SYS512
```

BUFFER POOL INFORMATION

Figure 19-1 (Cont). Memory Dump Example

```
 REAL   VIRTUAL  0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

 GLOBALLY SHAREABLE BOUND UNITS
 *************************

                        BOUND UNIT DESCRIPTION   03B0C3
 03B0C0/  0010 0000 0000 0005 0F03 0001 B801 0004  DEB3 0003 B2B8 0003 B0DB 0003 C3A3 0000  ................................
 03B0D0/  0000 0000 0000 0003 C3A3 0000 0000 0000  0000 0000 0000 4445 4255 4720 0000 C84D  ........................DEBUG ...M

                        BOUND UNIT ATTRIBUTES    03B0D8
 03B0E0/  0000 0000 0000 0003 C3A3 0000 0000 0000  0000 0000 0000 4445 4255 4720 0000 C84D  ........................DEBUG ...M
 03B0F0/  0000 0080 0000 0000 B801 0005 0000 0027  0000 0000 A901 0013 04EF 0888 0000 0000  ...............'................
 03B100/  1101 0026 0000 009F 0000 0000 1101 0028  0000 00FD 0000 0000 1101 002A 0000 012F  ...&...........(...........*.../
 03B110/  0000 0000 1101 002D 0000 0051 0000 0000  1101 002E 0000 00D8 0000 0000 1101 0030  .......-...Q...............0
 03B120/  0000 00CF 0000 0000 1101 0032 0000 009F  0000 0000 1101 0034 0000 004A 0000 0000  ...........2...........4...J....
 03B130/  1101 0035 0000 006D 0000 0000 1101 003E  0000 0062 0000 0000 1101 0037 0000 00C1  ...5...m.......>...b.......7....
 03B140/  0000 0000 1101 0039 0000 013B 0000 0000  1101 003C 0000 00C0 0000 0000 1101 003E  .......9...;.......<...........>
 03B150/  0000 0075 0000 0000 1101 003F 0000 013C  0000 0000 1101 0042 0000 0098 0000 0000  ...u.......?...<.......B........
 03B160/  1101 004A 0000 0057 0000 0000 1101 0045  0000 00C0 0000 0000 1101 0047 0000 00B7  ...J...W.......E...........G....
 03B170/  0000 0000 1101 0049 0000 0048 0000 0000  1101 004A 0000 008A 0000 0000 1101 004C  .......I...H.......J...........L
 03B180/  0000 006E 0000 0000 1101 004E 0000 0060  0000 0000 1101 004F 0000 0058 0000 0000  ...........N...`.......O...X....
 03B190/  1101 0050 0000 0135 0000 0000 1101 0053  0000 00C2 0000 0000 1101 0055 0000 00D0  ...P...5.......S...........U....
 03B1A0/  0000 0000 7101 0057 0000 062D 0000 0000  1101 0065 0000 0108 0000 0000 7101 0068  ....q..W...-.......e.........q..h
 03B1B0/  0000 00D8 0000 0000 7101 0068 0000 009C  0000 0000 1101 006E 0000 00DE 0000 0000  ........q..h...........n........
 03B1C0/  1101 0070 0000 00C3 0000 0000 1101 0072  0000 0045 0000 0000 7101 0073 0000 1C91  ...p...........r...E....q..s....
 03B1D0/  0000 0000 1101 00AE 0000 0109 0000 0000  1101 00B1 0000 00AC 0000 0000 7101 00B3  ...........................q....
 03B1E0/  0000 0107 0000 0000 1101 00B7 0000 00D8  0000 0000 1101 00B9 0000 0137 0000 0000  ...........................7....
 03B1F0/  1101 00BC 0000 0091 0000 0000 1101 00BE  0000 00A8 0000 0000 1101 00C0 0000 0129  ...........................)
 03B200/  0000 0000 1101 00C3 0000 0066 0000 0000  7101 00C4 0000 0118 0000 0000 1101 00C8  ...........f....q...............
 03B210/  0000 0082 0000 0000 1101 00CA 0000 0026  0000 0000 3101 00CB 0000 0128 0000 0000  ...............(....1.......(....
 03B220/  1101 00CF 0000 0133 0000 0000 1101 00D2  0000 0138 0000 0000 1101 00D5 0000 0133  .......3...........8...........3
 03B230/  0000 0000 1101 00D8 0000 0102 0000 0000  1101 00DB 0000 0130 0000 0000 3101 00DE  ...........................0....1...
 03B240/  0000 0087 0000 0000 1101 00E1 0000 00FE  0000 0000 1101 00E3 0000 010F 0000 0000  ................................
 03B250/  7101 00E6 0000 006A 0000 0000 3101 00E8  0000 00EC 0000 0000 1101 00EB 0000 00E5  q.......j....1.....................
 03B260/  0000 0000 1101 00ED 0000 013B 0000 0000  1101 00F0 0000 00AE 0000 0000 7101 00F2  ...........;...................q...
 03B270/  0000 00FD 0000 0000 1101 00F5 0000 0139  0000 0000 1101 00F8 0000 0087 0000 0000  ...........................9....
 03B280/  7101 00FA 0000 00D7 0000 0000 1101 00FD  0000 0025 0000 0000 1101 00FE 0000 013D  q...........................%.......=
 03B290/  0000 0000 1101 0101 0000 011A 0000 0000  1101 0104 0000 0112 0000 0000 1101 0107  ................................
 03B2A0/  0000 009A 0000 0000 1101 0109 0000 0108  0000 0000 1101 010C 0000 0078 0000 0000  ...........................x....
 03B2B0/  1101 0100 0000 0134 0000 0000 1101 0110  0000 013F 0000 0000 1101 0113 0000 003E  .......4...........?...........>
 03B2C0/  0000 0000 1101 0114 0000 002C 0000 0000  95C9 0020 0000 0000 FFC1 AD29 FFFF FFFF  ...........,..... ..........)....

                        BOUND UNIT:    DEBUG   03C3A3
 03C3A0/  0045 0000 0000 5A33 4558 4543 0000 0000  DFE7 EFE7 9FE7 EF67 1981 0096 ECF8 FFED  .E....Z3EXEC.............g........
 03C3B0/  ECC6 0034 08F8 0000 0DC6 0002 0901 008C  E8C6 0016 AFE7 8F67 0F67 8FE7 CF67 60C5  ...4.................g.g...g..
 03C3C0/  FFE2 0985 82C6 0007 8000 0505 3D01 090E  0F81 0068 3D01 0981 0065 AF67 CFE7 9852  ............m......h=....e.g...R
 03C3D0/  2C02 D5CE 00A7 CCF7 A877 ACC3 0008 0852  598e 82C2 0004 8000 0501 0054 586E 8756  ,.........W......RY.........TXn.V
 03C3E0/  ACC3 000C 80D2 0982 ACD3 CDC2 000A 0396  C8C4 FFFF 8AD6 0FFA 6891 ABC2 000F 2F06  ...........................h...../.
 03C3F0/  ABA2 E842 0002 688A 8256 ACC3 000A A892  ADD4 0284 8AD1 88D6 0FF9 ACC3 0008 3C0A  ...B..h..V..................<.
 03C400/  580F 9CCe 0031 8981 090B 3909 82F9 05FE  93C0 003F 19F7 D954 0929 0FF4 3C0A 9CC6  X....1....9.....?...T.)...<...
 03C410/  002D 5803 9CC6 0033 8981 0909 3907 82F9  05FE 93C0 002E 191A 0FF3 3C0A 98C0 FF84  .-X....3....9.............<.....
 03C420/  1C03 C85A C911 0983 19FD 0F8C 9CC6 002e  5803 9CC6 0035 8981 0905 3904 82F9 05FE  ...Z.............X....5....9.....
 03C430/  0F85 C87/ 8CF7 A85e 0F88 C877 8CF7 0FE7  03CE 004C 0CF7 A855 2883 9856 C894 D877  ...w...U...W.......L...U(..V...w
 03C440/  b877 ACF7 8751 E877 9CF7 ECF7 0CF7 8385  3002 8CC6 0037 8883 3042 8752 8751 C2EE  .w...Q.W.........0....7...0B.R.Q..
 03C450/  C1DF 0988 4D20 0903 2D06 027A 8751 C843  0003 8381 0F85 83C5 0001 0F10 0F0F 3D02  ....M ..-..z.Q.C..............=.
```

SYSTEM SHARABLE BOUND UNITS

Figure 19-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 1532:49.8      DUMPEDIT- 3.0-06/22/1713          GCOS6 NOD400-L3.0-04/02/1430  $$       PAGE 0058

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

**********************
*   POOL  $$    *
**********************


SHAREABLE BOUND UNITS
**********************


                        BOUND UNIT DESCRIPTION   04AEC3
        04AEC0/  0002 0000 0000 0000 0000 0001 8801 0004  D6F3 0004 AEEA 0008 AED8 0004 AE83 0000  ................................
        04AED0/  0000 0000 0000 0004 AE83 0000 0000 0000  0000 0000 0000 4147 5220 2020 0000 0000  ........................AGR     ....

                        BOUND UNIT ATTRIBUTES    04AED8
        04AED0/  0000 0000 0000 0004 AE83 0000 0000 0000  0000 0000 0000 4147 5220 2020 0000 0000  ........................AGR     ....
        04AEE0/  0000 0029 0000 0000 8801 0001 0002 0029  0000 0009 6102 0000 7006 0000 6101 0000  ...)............)....a.....a....

                        BOUND UNIT:    AGR     04AE83
        04AE50/  0002 0000 0000 0000 0000 ALF8 FFED C807  4002 0384 9870 1702 0F9D 0905 A842 0002  ................N....p........B..
        04AE90/  0F81 0001 9CL7 0003 L871 4984 9870 1712  0F91 4002 0904 9870 1704 0F8C A801 A970  .........qI...p....M....p........p
        04AEA0/  2453 0984 9870 1705 0F85 E070 0831 0001  0007 A851 0001 0103 0602 0000 0600 0000  $S...p.....p.....`.....0.............

                                                                                                                CURRENT MEMORY POOL
                                                                                                                SHARABLE BOUND UNITS FOR POOL $$
```

Figure 19-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 1532:49.8      DUMPEDIT= 3.0-06/22/1713          GC086 M00400-L3.0-06/02/1430  88  89   PAGE 0059

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

 TASK GROUP STRUCTURES FOR GROUP $$
 ******************************

 PERSON  IDENTIFICATION: OPERATOR
 ACCOUNT IDENTIFICATION: SYSTEM
 MODE    IDENTIFICATION: UPR
 BASE HARDWARE LEVEL:  0000
 NAME OF MEMORY POOL USED BY GROUP:  $$
 BIT MAP OF EXTERNAL SWITCHES: 0001
 NUMBER OF OUTSTANDING REQUESTS THIS GROUP HAS HAVE TO THE SYSTEM GROUP: 0000
 DIRECTORY DESCRIPTOR BLOCK FOR CURRENT_WORKING_DIRECTORY IS LOCATED AT: 0008A8      SIMPLE NAME IS  CYLON
 FILE CONTROL BLOCK FOR ERROR_OUT FILE IS LOCATED AT:  051433       SIMPLE NAME IS  CONSOLE
 FILE CONTROL BLOCK FOR  USER_OUT FILE IS LOCATED AT:  051433       SIMPLE NAME IS  CONSOLE
 SYSTEM TASK GROUP

 GROUP CONTROL BLOCK      0004A8
        0034A0/  0000 0000 0000 0000 0000 0000 c453 0000  0006 0004 A938 0000 0000 0003 1049 0003  ..............$8........8........I..
        0034B0/  0E59 0000 0502 0000 0001 0000 0000 0000  0000 0000 2424 AD31 BDA8 AD70 0000 C7A7  .Y..................$$.1...p....
        0034C0/  CAC1 0000 0000 AD3E 0000 0000 0BAB 0005  1433 0005 1433 0000 0000 0000 0000 0001  ........>..........3...3...........
        0034D0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        0034E0/  0000 0000 0000 0000 0005 00E0 0005 0B00  0000 0800 0000 0000 0000 0000 0820 083F  ..............................-.?
        0034F0/  0000 012D 0000 0000 0000 3191 0000 005A  0000 0001 0000 0000 0000 0000 0000 0005  ... .......1....Z................

 LOGICAL RESOURCE TABLE  03UE59
        03UE50/  00DA 1965 0870 0140 0870 000C 0FA1 FF47  0038 0000 D4BF 0000 0827 0000 F197 0000  ...e.p.M.p.....G.8........'.......
        03UE60/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 9F29 0000  ..............................)..
        03UE70/  9993 0003 9AA1 0000 98CF 0000 9CED 0000  9E08 0000 9875 0000 9FE0 0000 A0FE 0000  ..................u...........
        03UE80/  A21C 0000 0000 0000 A33A 0000 A3F1 0000  A4A8 0000 A55F 0000 A67D 0000 A79B 0000  .........:..........._...........
        03UE90/  4BC9 0000 AD96 0000 AEA6 0000 B04B 0000  B12D 0000 8281 0000 A9D7 0000 0000 0000  ..............K...-..............
        03UEA0/  C713 0000 C8A9 0000 CA07 0000 CB65 0000  DCC3 0000 DE21 0000 DF7F 0000 E0DD 0000  ..............e........!...........
        03UEB0/  E258 0000 E399 0000 E4F7 0000 E655 0000  ED28 0000 EE10 0000 EE4D 0000 EE8A 0000  .X...........U...+........M.......
        03UEC0/  E7B3 0000 E911 0000 EA6F 0000 EBCD 0000  0000 0000 0000 0000 301F 8010 301F 8020  .........o..............0...0.. .

 LOGICAL FILE TABLE       000502
        0034F0/  0000 012D 0000 0000 0000 3191 0000 005A  0000 0001 0000 0000 0000 0000 0000 0005  ... .......1....Z................
        003500/  0AC3 000F 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
```

TASK GROUP STRUCTURES

19-18

C205-00

Figure 19-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 15:32:49.8      DUMPEDIT- 3.0-06/22/1713          6C086 MD0400-L3.0-04/02/1430  88  88   PAGE 0060

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

 RESOURCE CONTROL TABLE INFORMATION FOR THE SYSTEM TASK GROUP


LOGICAL RESOURCE NUMBER: 0000
HARDWARE CHANNEL NUMBER: C00A
DEVICE IDENTIFICATION:   F200
DRIVER INDICATOR FLAGS:  0200
LAST DEVICE STATUS WORD: 0000
TASK CONTROL BLOCK ADDRESS:  00BF3E
ERROR LOG    BLOCK ADDRESS:
 RESOURCE CONTROL TABLE  00B4BF
        00B4B0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                   *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        00B4D0/  0000 0000 00BE 0000 1AFC 0000 04A4 0000  18F0 0000 BE41 BE61 4C45 0000 BF3E C00A  .....>..............A.aLE...>..

LOGICAL RESOURCE NUMBER: 0001
HARDWARE CHANNEL NUMBER: 0800
DEVICE IDENTIFICATION:   2305
DRIVER INDICATOR FLAGS:  1440
LAST DEVICE STATUS WORD: 0000
TASK CONTROL BLOCK ADDRESS:  0008B2
ERROR LOG    BLOCK ADDRESS:  000000
 RESOURCE CONTROL TABLE  000B27
        000700/  0008 BF00 0000 04A4 0000 0000 1461 0000  1B3C 0000 2A0A 0000 183C 01D5 001F 0002  ............¤...<..*....<.....
        000720/  E6AF 0000 1AA5 0004 BF58 0000 04A4 0025  0006 0000 0000 183C 0000 20F5 0000 11D9  .........¥.....X........<..*.....
        000740/  0000 105F 0000 BF3E 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ...._...>........................
        000760/  0000 0000 0000 0000 0000 0000 0000 0006  0000 0000 0000 06A1 0000 0F2A 0000 0D58  ...............................*...X
        000780/  0A17 0313 0000 0000 20F5 0000 000F 0000  0827 0217 0000 FFFE 0000 0000 0000 0000  ........".........'..............
        0007A0/  0000 0000 0000 8000 1308 0000 0882 0800  2585 1440 8000 0000 0000 0003 18FD 99CD  ..................#..@..............

LOGICAL RESOURCE NUMBER: 0002
HARDWARE CHANNEL NUMBER: 0025
DEVICE IDENTIFICATION:   0000
DRIVER INDICATOR FLAGS:  0000
LAST DEVICE STATUS WORD: 0001
TASK CONTROL BLOCK ADDRESS:  00F16A
ERROR LOG    BLOCK ADDRESS:
 RESOURCE CONTROL TABLE  00F197
        00F140/  0000 F16A 0025 0100 0000 0000 0000 0000  0000 5792 0000 FFFF 0703 0000 3586 6003  ...J.%.............W.........5.`.
        00F150/  0000 57F4 0000 01A0 0000 3584 0060 D4F9  0000 0000 0000 F144 0000 F139 0028 FC43  ..W.........5.....D.....9.(.C
        00F160/  0009 0000 07BA 0000 002C 0025 FF00 FF00  FF00 FF03 FF20 FF00 FF00 0000 0000 0000  .........:.....%.... ............
        00F170/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                   *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        00F190/  0000 0000 0000 0000 0000 0000 F16A 0025  0000 0000 0001 0A68 0000 A841 000F 6933  ............J.%.......h....A...i3

LOGICAL RESOURCE NUMBER: 000A
HARDWARE CHANNEL NUMBER: 0800
DEVICE IDENTIFICATION:   2380
DRIVER INDICATOR FLAGS:  1440
LAST DEVICE STATUS WORD: 0000
TASK CONTROL BLOCK ADDRESS:  0008B2
ERROR LOG    BLOCK ADDRESS:  000000
 RESOURCE CONTROL TABLE  009F29
```

RESOURCE CONTROL TABLE FOR THE SYSTEM TASK GROUP

19-19

C205-00

Figure 19-1 (Cont). Memory Dump Example

```
REAL   VIRTUAL  0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

                BOUND UNIT (TASK) NAME: AGR
                BOUND UNIT (TASK) RELOCATION FACTOR: 04AE85
                BOUND UNIT (TASK) DEFAULT START ADDRESS: 04AE85
                HARDWARE LEVEL: 0025
                LOGICAL RESOURCE NUMBER: FFFF
                BIT MAP OF ENABLED TRAP NUMBERS: 0000000000000000
                RESERVED OVERLAY AREA IS LOCATED AT: 000000
                CURRENT  OVERLAY AREA IS LOCATED AT: 000000
                FILE CONTROL BLOCK FOR COMMAND_IN FILE IS LOCATED AT:  000000      SIMPLE NAME IS
                FILE CONTROL BLOCK FOR    USER_IN FILE IS LOCATED AT:  000000      SIMPLE NAME IS
                COMMAND

                TASK CONTROL BLOCK      04A938     AGR
044BE0/ 0005 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0004 A938 0025 0000 0006  ..............................8.%.....
044BF0/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0004 A8F8 0005  ................................
044900/ 13E3 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
044910/ 0000 0000 0000 0000 0116 001b 1180 0006  0000 0000 0000 0000 0004 AEC3 0004 AE85  ..............................
044920/ FFFF FFFF FFFF 0004 A898 0000 04A4 0000  0004 A927 0004 A927 0000 0000 0004 A938  ..................'...'........8
044930/ 0025 0100 0000 0003 0FC3 0000 0000 5D42  0000 FFFF B703 0000 3586 6003 0004 AC9A  .%..............]B........5.`.....
044940/ 0000 01A0 0000 3584 0000 0000 0000 0000  0004 A932 0004 A927 0000 B18A C804 003A  .......5.............2...'.......:
044950/ 0002 0000 030F 0025 FF00 FF00 FF00 FF03  FF20 FF00 FF00 0000 0000 0000 0000 0000  .......%..... ..... ..... .....
044960/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
044970/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                TRAP SAVE AREA      005D42
                INSTRUCTION WHICH TRAPPED IS AN MCL AT LOCATION  002FAF.  FUNCTION CODE IS 0103.
                INSTRUCTION: 0001    P_COUNTER: 002FB1   I': 3F20   Z: 80C1      A: 002FAF  R3: 000F   B3:006708
005D40/ 0060 2495 0000 6702 3F20 000F 0001 80C1  0000 2FAF 0000 2FB1 0000 6708 0004 ABE9  ..$...g.? ......../.../...g.....
005D50/ 0000 676A 0000 01A0 0000 04A4 0000 0335  003A 0002 FF08 0003 0000 0000 0000 0000  ..gj...........5.:..............
005D60/ 0000 0000 0000 0000 0000 0000 0000 0000  0004 A4C3 0000 0000 0000 0000 0001 0000  ................................
005D70/ 0000 030F 0000 0000 0000 04A4 0000 04A4  0000 0000 0000 0030 3033 3046 0000 0000  ...................0030F....
005D80/ 3030 3030 3030 0000 0000 0012 0012 4141  4752 3A20 3137 2030 3330 4620 2832 3529  000000........AAGR: 17 030F (25)
005D90/ 0000 04A4 B18A C804 003A 0002 0802 030F  0000 2807 0000 0004 04F8 0000 04A0 0004  .........:........(.............
005DA0/ A938 C804 B18A 003A 0000 20D3 0000 20DA  0000 20D0 0000 0000 3F01 005E 0001 80C1  .8.....:..-...-...-.....?..^....

                MCL WORK SPACE      04ABE9
044BE0/ 0006 0000 0000 0000 0000 0000 0000 0004  AC0F 00FC 0000 0000 0000 0005 0000 0000  ................................
044BF0/ 030F 0000 0000 0000 04A4 0000 04A4 0000  0000 0000 0030 3033 3046 0000 3031 3030  .....................0030F..0100
044C00/ 3030 3030 0000 0000 0035 0014 4153 5447  3A20 3137 2030 3330 4620 2832 3529 2020  0000.....5..ASTG: 17 030F (25)
044C10/ 5245 4645 5245 4E43 4520 544F 2055 4E41  5641 494C 4142 4C45 2052 4553 4F55 5243  REFERENCE TO UNAVAILABLE RESOURC
044C20/ 4545 2052 4553 4F55 5243 4563 0005 0000  1631 0001 0000 0000 1559 0004 AE80 0000  EE RESOURCEe.....1.....Y.......
044C30/ 04A4 0005 0005 05C3 0000 49E2 0005 05C3  0004 0000 2A0A 0000 0000 2A0A 0000 41D9  ..........I.........*.....*...A.
044C40/ 0005 1103 0005 1203 0005 17D3 0030 0000  2A0A 0000 4109 0005 1423 0000 9807 0005  ..............0...*...A.....#.....
044C50/ 0000 4941 0000 0005 004A 0000 3807 0005  1423 0000 3A90 0000 4AA6 0000 388A F029  ..IA.....J..8....#...:...J...8..
044C60/ 0000 35C6 0004 A4E6 0004 A4E6 0000 0000  4941 0000 003F 0000 1631 0001 0000 0000  ..5.............IA...?...1.......
044C70/ 1559 0004 A4C0 0000 04A4 0035 0000 3643  0000 1B3C 0116 0016 0002 F2AD 431A 0000  .Y.........5...6C...<.........C...
044C80/ 0000 0000 0000 2666 0004 D4F8 0000 04A4  B1BA C804 003A 0002 0802 030F 0000 2807  ......(f.............:........(.
044C90/ 0000 0004 04F8 0000 04A4 0004 A938 C804  B1BA 003A 0000 20F5 0000 20DA 0000 20D0  .............8.....:...-...-...-.
044CA0/ 0006 0000 0000 0000 0000 0005 0F83 0000  0000 1000 0000 0000 0000 0000 0001 0000  ................................
```

TASK STRUCTURES FOR AGR

Figure 19-1 (Cont). Memory Dump Exampl

```
 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

CHAIN OF WORK SPACE BLOCKS FOR THE CURRENT TASK GROUP
**********************************************************

                WORK SPACE BLOCK        0500E0
0503E0/  0001 0005 0660 0005 00EF 0000 95C9 00C5  0001 0004 CA2D 0005 0438 0000 0000 0000  ....................-...8.......
0503F0/  0000 0001 FE02 0000 0000 0001 36C0 0004  CA23 0000 0000 0030 DBF9 0000 0000 2020  ..............6.....#.....0.......

                WORK SPACE BLOCK        0506C0
0505C0/  0001 0005 0660 0005 06CF 0000 9564 00C5  0001 0004 EBAD 0005 0438 0000 0000 0000  ....................d.....8.......
0505D0/  0000 0001 FE02 0000 0000 0004 FAC3 0004  EBA3 0000 0000 0100 B0A1 0000 0000 2020  ..............0.....1.....  ....

                WORK SPACE BLOCK        0500C0
0503C0/  0001 0005 0660 0005 00CF 0000 8FF6 00C5  0001 0004 CB2D 0005 0438 0000 0000 0000  ..... ...............-...8.......
0503D0/  0000 0001 FE02 0000 0000 0001 3503 0004  CB23 0000 0000 574F 5248 5320 5354 5220  ..............5.....#.....WORKS STR

                WORK SPACE BLOCK        0506D0
0506D0/  0001 0004 FAC0 0005 062F 0000 9719 00C5  0001 0004 C9ED 0005 0438 0000 0000 0000  .........../...............8.......
0506E0/  0000 0001 FE02 0000 0000 0004 07C3 0004  C9E3 0000 0000 2020 2020 2020 2020 2020  ....................  ....

                WORK SPACE BLOCK        04FAC0
04FAC0/  0001 0005 0A60 0004 FACF 0000 6616 00C5  0001 0004 AB0D 0005 0438 0000 0000 0000  .....`..................8.......
04FAD0/  0000 0001 FE02 0000 0000 0001 3719 0004  AB03 0000 0000 0000 0000 0000 0000 0000  ..............7.....................

                WORK SPACE BLOCK        050A60
050A60/  0001 0004 C000 0005 0A6F 0000 609F 00C5  0001 0005 032D 0005 0438 0000 0000 0000  .........`.....o.....-...8.......
050A70/  0000 0001 FE02 0000 0000 0001 34AA 0005  0323 0000 0000 0100 E3D5 0000 0000 2020  ..............4.....#.....  ....

                WORK SPACE BLOCK        04CD00
04CD00/  0003 0004 F7C0 4120 4449 5220 2020 2020  2032 3536 2020 2020 2020 3230 2020 2020  ......A DIR      256      20
04CD10/  2020 3230 2020 2020 2030 0000 0000 0000  0048 0001 0001 0003 0003 5359 5332 3536    20      0.......H.........SYS256
04CD20/  0022 0032 0100 0032 0001 5359 5335 3132  0022 0032 0200 0032 0001 4449 5220 2020  ."..2...2..2...2..DIR
04CD30/  01C0 0014 0100 0014 0000 0000 0000 CD34  0004 CD34 0004 CD30 0000 0000 0001 0001  . ..............4....4....4........
04CD40/  0004 0000 0004 0000 0000 0001 0001 0000  0000 0000 0100 0000 0602 0000 0000 0000  ................................
04CD50/  0000 0100 0000 0000 0000 0000 0000 0000  0000 0000 0000 0059 00AE 0000 0801 0100  ...........................Y.........

                WORK SPACE BLOCK        04F7C0
04F7C0/  0001 0004 FCE0 0004 F7CF 0000 9165 00C5  0001 0004 D54D 0005 0438 0000 0000 0000  ....................N...8.......
04F7D0/  0000 0001 FE02 0000 0000 0000 62AC 0004  D543 0000 0000 2020 2020 2020 2020 2020  ................C....

                WORK SPACE BLOCK        04FCE0
04FCE0/  0002 0005 B8A0 0000 82AC 0081 0011 0000  0000 0000 3031 0000 0000 2300 0000 0000  ....................01....#.....
04FCF0/  0000 0000 0000 0000 0000 0100 8001 0000  0000 0001 0000 0000 0000 0000 0000 0000  ................................
04F600/  0000 0000 0000 0000 2454 472E 2442 2020  2020 2020 0000 0000 0000 0000 0000 0000  ........$TG.$B  ............
04F610/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                WORK SPACE BLOCK        0300A0                                                                  WORK SPACE BLOCKS USED BY SYSTEM GROUP $$
```

Figure 19-1 (Cont). Memory Dump Example

```
REAL   VIRTUAL  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F        ABCII REPRESENTATION

****************
*   POOL L2    *
****************


SHAREABLE BOUND UNITS
*********************

                        BOUND UNIT DESCRIPTION   04C583
        04C580/  0002 0000 0000 0005 12E3 0001 A801 0004  C4B5 0004 C580 0004 C590 000A 0000 0000  ................................
        04C5A0/  0000 0000 0000 000A 0000 0004 E0A3 0000  0000 0000 0000 4142 4252 4556 0000 C000  ......................ABBREV....

                        BOUND UNIT ATTRIBUTES    04C5B0
        04C5C0/  0000 0000 0000 000A 0000 0004 E0A3 0000  0000 0000 0000 4142 4252 4556 0000 C000  ......................ABBREV....
        04C5E0/  0000 05C0 0000 0000 A801 0001 0060 0C3A  0000 0000 A801 0007 0000 05CD 0000 0000  ................:..............

                        BOUND UNITS:    ABBREV  0A0000
0582C0/ 0A3C00/  C0C0 04F2 0001 0A00 1981 04DF A870 0001  0001 0A01 1981 04D9 A870 0031 0001 0A01  ..............p.....p.1.....
0582E0/ 0A3C20/  1981 0493 A870 0005 0001 0A01 1981 04C0  4302 C0C6 006E 0001 1408 C0C6 0085 0001  .....p.........C....n..........
0583C0/ 0A3C40/  1400 2C00 0004 1C2E 9AC3 0004 00FF 92EF  1D2E 09FE 1C50 97EF 1C46 97EF 1C20 97EF  ..,.....................P...F... ..
0583A0/ 0A3C60/  AF50 A093 2C00 00C0 0006 3C00 92FF 1020  09FE 1C3E 97F8 8AD3 0008 69FF C0C6 0005  .V..,.....n<.... ...>.......i.....
0583C0/ 0A3C80/  0001 1020 9970 0209 099E C0C6 0010 0001  1030 1981 01E7 9846 0010 9F46 002E 9F46  ... .p.............0.....F...F...F
0583E0/ 0A3CA0/  002E C0C0 002E 0001 1050 1981 0108 0001  1055 1981 0107 C0C6 0028 0001 1025 1981  .+........P........U........+...X..
0584C0/ 0A3CC0/  0101 0FC0 1981 01CE 9846 0005 9F46 002E  9F46 0028 C0C6 002E 0001 1050 1981 01C2  .........F...F...F.+........P....
0584A0/ 0A3CE0/  8E40 0003 0002 0501 0473 8846 006E 0001  0501 FFF7 82C6 0068 0260 0581 004A 0001  .F.k.....s.F.k.........k.`...J..
0584E0/ 0A3D00/  0506 82C6 0068 0040 0506 82C6 006E 0020  0515 0FA6 9846 0067 1EE8 82C6 0068 0200  .....k.@.....k. .....F.g.....k..
0585C0/ 0A3D20/  0582 1EF3 1041 8ED1 C0C6 003E C894 5C14  0670 0018 0001 0504 0F93 9846 0067 1EF6  .....A.....>..\..p.........F.g..
0585A0/ 0A3D40/  82C6 0063 0200 0582 1EF5 1041 8ED1 C0C6  003E C894 5C09 0870 0009 0001 0505 82C6  ...k.........A.....>..\..p.........
0585E0/ 0A3D60/  0068 0200 0596 C846 0094 D846 0095 2C00  8D46 0094 8405 9846 0067 1EF6 1041 8ED1  .k.....F...F.....F.....F.g...A...
0586C0/ 0A3D80/  C0C6 003E C894 5C09 0870 0009 0001 0505  ACF8 FFED 82C2 0004 0040 0596 8846 0068  ...>..\..p.............@...F.k
0586A0/ 0A3DA0/  0400 0509 9846 006A 9570 003F 1D12 0903  101A 096A C8C6 003E E846 0067 7C00 0001  .....F.j.p.?.............>.F.g....
0586E0/ 0A3DC0/  0903 1981 04D2 A8C0 0424 2C00 88C6 011D  3C00 E870 0100 0000 69FF C8C6 011D E870  .........s.....<..p....i.......p
0587F0/ 0A3DE0/  0100 7C00 0001 0802 8D46 0069 9970 021F  0901 0045 1981 03E9 0001 0506 8D46 0094  .........F.i.p.....E.........F..
                                                  ***   REFER TO PHYSICAL DUMP   ***
0587C0/ 0A3E00/  4647 4849 4A4B 4C4D 4E4F 5051 5253 5455  5657 5859 5A7B 7C7D 7E7F 2041 5247 5320  FGHIJKLMNOPQRSTUVWXYZ.....-ARGS

                        BOUND UNIT DESCRIPTION   0512E3
        0512E0/  0002 0000 0000 0004 F363 0003 8801 0005  0ED3 0005 1310 0005 12F8 000A 0000 0000  .........c......................
        0512F0/  0000 0000 0000 000A 0000 0005 11B3 0000  0000 0000 0000 414D 5420 2020 0000 0001  ......................AMT    ....

                        BOUND UNIT ATTRIBUTES    0512FB
        0512F0/  0000 0000 0000 000A 0000 0005 11B3 0000  0000 0000 0000 414D 5420 2020 0000 0001  ......................AMT    ....
        051300/  0000 00E9 0000 0000 8801 0001 0000 00E9  0000 0000 1001 0003 0014 0105 0000 0000  ................................
        051310/  0000 0100 0000 0000 0000 0000 0000 0000  0000 8010 0090 0001 0000 0000 0000 0000  ................................

                        BOUND UNITS:    AMT     0A0000
055F00/ 0A3000/  EC80 0000 0018 9846 0040 0001 0000 8943  FFFF 00C0 D385 9870 0C03 BCF8 FFED BCC3  .......F.@.....C.......p........
055F10/ 0A3010/  0030 8D03 0904 89C3 FFFB 0988 F870 006E  6C00 2C00 0001 0402 190C 8870 0017 F851  .0.............p.n!.........p...Q
055F20/ 0A3020/  E870 0000 0001 0F00 A870 FFFF 0001 0103  9884 B857 CBC0 008A 0001 0A00 A870 0000  .p.........p.........M.........p..
```


Figure 19-1 (Cont). Memory Dump Example

```
 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ASCII REPRESENTATION

 TASK GROUP STRUCTURES FOR GROUP L2
 **********************************


 PERSON IDENTIFICATION: EISENBERG
 ACCOUNT IDENTIFICATION: MP3
 MODE   IDENTIFICATION: 601
 BASE PARTITION LEVEL: 0025
 NAME OF IMPURE POOL USED BY GROUP: L2
 BIT MAP OF EXTERNAL SWITCHES: 0000
 NUMBER OF OUTSTANDING REQUESTS THIS GROUP HAS MADE TO THE SYSTEM GROUP: 0002
 DIRECTORY DESCRIPTOR BLOCK FOR CURRENT_WORKING_DIRECTORY IS LOCATED AT: 04B6E3      SIMPLE NAME IS  LINK
 FILE CONTROL BLOCK FOR ERROR_OUT FILE IS LOCATED AT:  0601F3      SIMPLE NAME IS  TTY03
 FILE CONTROL BLOCK FOR USER_OUT FILE IS LOCATED AT:  0601F3      SIMPLE NAME IS  TTY03


 GROUP CONTABLE BLOCK      060BL3
        04D6C0/  0003 0000 0000 0004 8523 4C32 0025 0044  0003 AD05 0003 AAC5 0003 1073 0006 00A4  .........#L2.%.D.............
        04D6D0/  0006 4127 0002 0001 0000 0000 0000 0000  0000 4C32 6DA7 6E5E 6F03 0000 8E1F 0000  ...'..............L2m.n^o.........
        04D6E0/  0C00 0000 0770 0000 0000 86E3 0006 01F3  0006 01F3 0006 01C3 0009 88E8 0000 0000  .....P..........................
        04D6F0/  0000 0006 0000 0004 F4E9 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        04D700/  0000 0000 0000 0008 0700 0008 0340 203C  8800 0000 0000 0000 0000 0000 0000 0000  ..............@(<................
        04D710/  055E 0000 1F01 0000 146A 0000 0108 0000  0000 0000 0000 0000 0000 0000 0000 0000  .^.........j....................


 LOGICAL RESOURCE TABLE  060GA4
0579A0/ 0603A0/  0007 0000 0000 003C 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  .......<........................
0579B0/ 0603B0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                  *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
057A10/ 060410/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0001 0000  ................................


 LOGICAL FILE TABLE      060427
057A10/ 060410/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0001 0000  ................................
057A20/ 060420/  0000 0000 0000 0000 0000 0000 0028 0000  0000 0000 0000 0000 0000 0000 0000 0000  ............(...................

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```


Figure 19-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP      1982/06/28 15:32:49.8      DUMPLOIT- 3.0-06/22/1713          GCU36 MOD400-L3.0-04/02/1430  L2  L2   PAGE 0101

 REAL    VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

        0346F0/  0000 0000 0000 0000 0004 CEC3 0000 0000  0004 C863 0000 0000 0005 0843 0005 0863  ..................c........C...c
        0347A0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        0348A0/  0300 1400 0003 8521 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 000D  ........!.......................
        0349F0/  0000 0000 0000 0000 0000 0003 ACFF 0003  ACF4 0007 0026 0000 000B 0000 0000 0000  ........................&.........
        034AC0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0003 ACC8 0006 0223 0000 0000  ...........................H.....
        034B00/  0000 0000 0000 0000 0000 0006 0193 0006  0195 0006 0193 0060 0000 0000 0000 000A  ...........................`........
        034BE0/  025E 0000 0000 9479 8DC7 0004 0BC3 0000  0000 0006 0303 000A 0000 0000 0000 FFFF  .^.....y.......................
        034CF0/  0003 00E5 0000 D0C3 0000 0003 ACF4 0003  ACF4 0007 0000 0003 AD05 0026 0300 0020  ...........................&...
        034D00/  0003 AL40 0000 0000 090A 0000 FFFF 8703  0000 3586 6003 0000 696C 0000 81AD 0000  ....F.....`.....5.`...il........
        034D10/  3504 0003 1AL3 0000 0000 0003 ACFF 0003  ACF4 0000 431A 0000 0007 0000 0000 0000  5.............C.................
        034D20/  0026 FF00 FF00 FF00 FF03 FF20 FF00 FF00  0008 04A9 0000 0000 0000 0000 0000 0000  .&.......... ...................
        034D30/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        034D40/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                         TRAP SAVE AREA      006900
                INSTRUCTION WHICH TRAPPED IS AN MCL AT LOCATION  002FAF.  FUNCTION CODE IS 0103.
                INSTRUCTIONS: 0001    P_COUNTER:  0V2FB1   I's 3F24    Z: 80C1      A: 002FAF   R3: 000F    B3:006430
        0D5900/  0005 0003 0000 66A8 0004 FA9E 000C 0004  0004 000C 0000 642A 3F24 000F 0001 80C1  .....h..............d*?$........
        0D5910/  0000 2FAF 0000 2FB1 0000 6430 0000 692C  0000 6492 0000 01AD 0004 DBC3 00C9 0008  ../.../...d0..i,..d.............
        0D5920/  0007 0000 FF00 0003 0004 000D 0000 0000  0002 0008 FFFF FF00 0000 0000 0000 0000  ................................
        0D5930/  0008 1403 0000 2000 0000 0000 0009 0000  0000 030F 0000 0000 0000 0000 0000 0000  ...... ........................
        0D5940/  0000 0000 0000 0030 3033 3040 0000 0000  3030 3030 3030 0000 0000 0003 AAC5  .......0030F....000000..........
        0D5950/  0000 8LC7 0004 D0C3 0000 431A 0000 0000  2034 0000 0000 8DC9 0000 8687 0008 282A  ..........C.....+4..............(*
        0D5960/  1006 0003 1AL3 0003 AAC5 0004 0BC3 0003  AD05 0000 431A 0007 0000 2DF5 0000 2D9A  ....................C.....5...-.
        0D5970/  000C 2060 0000 0000 3F20 FFFF 0001 8001  000A 0142 000A 0144 0008 0042 0000 6994  ..-.....? .........B...D...B..i.

                         TRAP SAVE AREA      006ACA

***  VIRTUAL ADDRESS IS INVALID:   1D0000

***  THE PAV VIRTUAL ADDRESS IS AT  006434
                INSTRUCTION: 8F87    P_COUNTER:  0A2F93   I's 3104    Z: 0002      A: 0803C4   R3: 0000    B3:1D0000

***  VIRTUAL ADDRESS IS INVALID:   C9000H

***  THE PAV VIRTUAL ADDRESS IS AT  006492
                POSSIBLE CONTEXT
                  R1, R2, R4, R5, R6, R7:  1060  0000  0000  0007  0002  0010
                  B1, B2, B4, B5, B6, B7:  C9000B  00000V  0B03C8  0A145B  080398  080380
        0D5420/  0004 0F20 0000 63C8 0000 6C04 4000 0002  FFFF 0000 0000 0000 3104 0008 8F87 0002  ...{..c...l.d.............1.......
        0D5430/  0008 03C4 000A 2F93 0010 0008 0000 644L  0008 0380 0008 0398 000A 145B 0008 03CB  ....../.......dL.............{....
        0D5440/  0000 0000 00L9 0008 0004 0010 0002 0007  0000 0000 1060 FF00 0000 0000 0000 0000  ..................-.............
        0D5450/  0008 1403 0000 0000 000C 0000 0001 000C  0000 FFFF 0000 8000 0000 0000 49C5 0000  ..............................I...
        0D5460/  0009 000B 0372 0010 8DU9 00FF 0060 0000  0000 0026 0000 4539 0003 1CA3 0000 00C   ....r..............&..E9.........
        0D5470/  0000 000C 41C1 0098 000B 0002 53C4 0005  0000 0000 1631 0001 0000 0000 1559 000+  ....A........S.........1........Y..
        0D5480/  0D30 0000 0000 04A4 0000 0000 0000 0000  24A9 0008 09B1 0000 642A 0000 0000 0000  ................$.........d*........
        0D5490/  0000 0000 0000 0000 0000 3F13 2020 0001  80C1 000A 0083 000A 0085 0004 CFE9 0000  ..........?.  ..................
```

```
                         DOUBLE UNIT DESCRIPTION   060304
057C00/ 063500/  0002 0000 0000 0000 0000 0001 2101 0006  0000 0006 032A 0006 031B 000A 0000 0000  .............!.........*..........
057C10/ 063510/  0000 0000 0000 000A 0000 0005 0843 0000  0000 0000 0000 5359 5353 4156 0000 0000  ..............C.........SYSSAV....
```


Figure 19-1 (Cont). Memory Dump Example

REAL VIRTUAL 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII REPRESENTATION

```
BOUND UNIT (TASK) NAME:  SYSSAV
BOUND UNIT (TASK) RELOCATION FACTOR: 0A0000
BOUND UNIT (TASK) DEFAULT START ADDRESS: 0A0000
HARDWARE LEVELS  0026
LOGICAL RESOURCE NUMBERS: FFFF
BIT MAP OF ENABLED TRAP NUMBERS: 0060000000000000
RESERVED OVERLAY AREA IS LOCATED AT: 000000
COMMENT  OVERLAY AREA IS LOCATED AT: 000000
FILE CONTROL BLOCK FOR COMMAND_IN FILE IS LOCATED AT:  066193      SIMPLE NAME IS  TTY03
FILE CONTROL BLOCK FOR    USER_IN FILE IS LOCATED AT:  066193      SIMPLE NAME IS  TTY03
COMMAND_IN FILE IS INTERACTIVE
```

SEGMENT DESCRIPTORS

SEGMENT	BASE	SIZE	READ	WRITE	EXECUTE
6.0	057900	000900	3	0	3
6.0	06A300	001800	3	3	3
A.0	05L200	003800	3	3	3
B.0	07L200	002100	3	3	0

```
          SEGMENT  6.0 MEMORY CONTROL BLOCK     04CEL3    6DS
04?cC0/ 0003 0000 0000 2000 0004 0004 0000 0000  0000 0000 0009 0001 0000 0000 0602 0579 ..... .....Y
04?cD0/ 3066 0001 0000 0000 0000 0000 8A38 0001  0003 0000 0000 0000 0000 0000 0000 0000 0.....8.....
04?cE0/ 0000 0000 0001 0000 0000 0000 0394 0001  0004 CEE2 0003 AA06 0000 0000 0005 0000 ..... .....

          SEGMENT  6.0 MEMORY CONTROL BLOCK     04C863    GWS
04?c60/ 0003 0000 0000 2006 0004 0004 0000 0002  0000 0000 001B 000A 0000 0000 0802 06A3 ..... .....
04?c70/ 0014 0001 0000 0000 0000 0000 93A7 0001  0003 0000 0000 0000 0000 0000 0000 0000 ..... .....
04?c80/ 0000 00C0 0001 0000 0000 0000 0014 0001  0004 C882 0003 AA06 0000 0000 0005 0006 ..... .....

          SEGMENT  A.0 MEMORY CONTROL BLOCK     050843    BU
053c40/ 0001 0000 0000 2010 0001 0001 0000 0000  0000 0000 0038 0000 0000 0000 0A00 05B6 ..... .....8.....
053c50/ 0037 0001 0000 0000 0000 0000 82EB 0001  0001 0000 0000 0000 0000 0000 0000 0000 .7..... .....
053c60/ 0001 0000 0000 2020 0001 0001 0000 0000  0000 0000 0021 0000 0000 0000 0800 07E2 ..... .....!.....

          SEGMENT  6.0 MEMORY CONTROL BLOCK     050863    USER
053c60/ 0001 0000 0000 2020 0001 0001 0000 0000  0000 0000 0021 0000 0000 0000 0800 07E2 ..... .....!.....
053c70/ 0C20 0001 0000 0000 0000 0000 6B0A 0001  0001 0000 0000 0000 0000 0000 0000 0000 . ..... .....
053c80/ 0002 0000 0000 0000 0000 0005 04b9 0001  0001 0051 0008 034A 4C4F 474F 5554 0000 ..... .....Q....JLOGOUT..

          TASK CONTROL BLOCK     03A005    SYSSAV
03AC40/ 0009 0000 0000 0003 A005 0026 8000 301F  8010 301F 8020 301F 8030 301F 8040 301F .....8...0...0... 0...00..00.
03AC50/ 8050 301F 0060 301F 8070 301F 8080 301F  8090 301F 80A0 301F 80B0 301F 80C0 301F .P0..`0...p0...0...0...0...0...0.
03AC60/ 80D0 301F 80E0 301F 80F0 301F 8100 31FF  8200 31FF 8300 31FF 8400 3117 8500 3017 ..0...0...0...1...1...1...1...0.
03AC70/ 0579 3005 0000 0000 66A3 001A 0000 0000  8586 0037 87E2 0C20 0000 0000 0000 0000 .Y0.....f.....7... .....
```

TASK STRUCTURES FOR SYSSAN (CONT)

Figure 19-1 (Cont). Memory Dump Example

- Batch Group Data (shown if batch group is present)

  - Virtual address of beginning of background
  - Virtual address of the end of background.

- Memory Pool Data*

  - Pool identification
  - Starting address of pool
  - End address of pool
  - Total size of pool
  - Physical start address
  - Total available space
  - Maximum contiguous available space
  - Number of available fragments (pieces) of pool space
  - Number of users
  - Table of attributes for each pool.

- Additional pool information

  - Memory pool descriptor
  - Bit map (unless it is a queue-managed pool)
  - Segment descriptors.

- System Symbol Table

  The names and values of all symbols that have an entry in the system symbol table are displayed. Symbols are grouped according to the bound unit(s) in which they occur.

- File System Structures

  - Record locking pool control block
  - Volume descriptor blocks (VDBs)
  - Directory descriptor blocks (DDBs)
  - File descriptor blocks (FDBs)
  - Currency control blocks
  - Remote extent blocks
  - Wait control blocks
  - User control blocks
  - Semaphore control blocks
  - Record locking control blocks
  - Device descriptor blocks (DDBs)
  - Buffer control blocks
  - Public buffer pool headers (BPHs)
  - Buffer control blocks (BCBs)
  - Buffers.

*Supplied for each memory pool. The pool name for the batch group is BATCH.

The hierarchy of these file system structures is indicated by the dump as shown in Figure 19-2, which is an abridged section of a logical dump. Each block is assigned an integer that corresponds to the level of the block in the hierarchy. The headings of all blocks are indented according to the depth of the block. This makes it easy to see which files belong to volume major directories and which belong to subordinate directories. The display of the tree of file system structures may be suppressed by the -NF argument.

The following file system structures are also displayed:

- Free indirect request block queue (only when editing a dump file)
- Globally sharable bound units
  - Bound unit description
  - Bound unit attributes
  - Bound unit.

TASK RELATED INFORMATION

The preceding logical dump information is obtained from the operating system area of memory and occurs once within a logical dump. The following information can be repeated more than once depending on the number of active pools, task groups, and tasks. This information is presented in the following order:

1. Memory pools (as allocated at CLM time) if there are task groups assigned to them.
2. Task groups within a memory pool.
3. Tasks within a task group.

Memory Pool Structures

The following information is repeated for each pool with assigned task groups:

- Sharable Bound Units
  - Bound unit description
  - Bound unit attributes
  - Bound unit.

Task Group Structures

The following information is repeated for each task group in a pool.

- Edited Task Group Information

  - User name, account, and mode
  - Assigned memory pool
  - Bit map switches
  - Outstanding requests to system group
  - Address and name of control block for current working directory, error-out, and user-out

- Group control block

- Logical resource table

- Logical file table

- Task structures (detailed below)

- File control blocks (if there are active files)

- Work space blocks.

NOTES

1. For the system task group, IRBs (and hence also RBs) are displayed only when DPEDIT is processing a dump file; i.e., the display is suppressed when the input is from current main memory.

2. Work space blocks and FCBs for the batch task group are not displayed when the batch group is rolled out.

Task Structures

The following information is repeated for each task in a group:

- Edited Task Information

  - Bound unit name, location, and start address
  - Hardware level
  - Logical resource number
  - Enabled trap bit map
  - Reserved and current overlay area locations
  - Control block name and address for user-in and command-in

- Segment descriptor table (swap pool only)
- Memory control block for each segment (swap pool only)
- Task control block
- Trap save area
- MCL word space (for an MCL trap)
- Bound unit description
- Bound unit attributes
- Bound unit
- Overlay areas (if an overlay area table was used).

The firmware-defined fields (instruction, P-counter, I', Z, A, R3, and B3) for each trap save area (TSA) are displayed. If the instruction is a monitor call, the function code is also displayed.

In addition, a possible context of the remaining data and address registers (R1, R2, R4, R5, R6, R7, B1, B2, B4, B5, B6, and B7) is displayed for each trap save area. This context, which is extracted from the work space area of the trap save area, may not be valid in all cases, but in general, is correct due to internal conventions of the Executive.

DPEDIT Command

The DPEDIT command loads the Dump Edit utility program. Immediately after Dump Edit begins executing, a message is issued to the error-out file giving the unique version number in the following format: DPEDIT-nnnn-mm/dd/hhmm. The message "DUMP COMPLETE" is issued to the error-out file immediately before the execution of Dump Edit terminates. The format for the DPEDIT command is:

```
DPEDIT [path] [ctl_arg]
```

ARGUMENTS:

path

Pathname of the memory dump file to be printed. Either the path argument or the -MEMORY control argument must be specified.

ctl_arg

Control arguments; zero, one, or more of the following control arguments may be entered, in any order:

{-SWAP_FILE swapfile name
-SF swapfile name}

Identifies the swapfile (if any) associated with the dumpfile specified by the path argument. The swapfile name can be a simple name or pathname.

This argument is used only when dumping a file previously produced by MDUMP. The argument allows DPEDIT to include in its dump information about task groups that were swapped out of the swap pool at the moment recorded by the MDUMP image. If the DPEDIT command requests information about a task group that was swapped out and -SWAP_FILE is not specified, an error results.

{-NO_LOGICAL
-NL}

No logical dump of system control structures produced.

Default: Logical dump produced.

{-NO_PHYSICAL
-NP}

No physical dump of memory produced.

Default: Physical dump produced.

{-FROM X'address'
-FM X'address'}

Low-memory address of area that will appear in physical dump; must be specified in hexadecimal. The specified address must be a virtual address if memory is being processed, and a physical address if a dump file is being processed.

Default: Absolute 0.

-TO X'address'

High-memory address (up to five hexadecimal digits) of area that will appear in physical dump; must be specified in hexadecimal. The specified address must be a virtual address if memory is being processed, and a physical address if a dump file is being processed.

Default: High memory address of the dump file.

{-MEMORY | -MEM}

Produces a dump of main memory. If both the path argument and this argument are specified, an error message appears at the terminal. If the -FROM (and/or -TO) control argument is used in conjunction with the -MEMORY control argument, then the address that is specified must be a virtual address.

Default: A dump is produced of the file specified in the path argument.

{GROUP | -GP} group id [group-id] ...

Requests the logical dump to contain task group-related information for the specified group(s) only.

Default: Task group information for all groups is included in the logical dump.

{-NO_FILES | -NF}

No tree of file management structures is produced.

Default: A tree of file management structures is produced.

-ME

Dump only the group in which DPEDIT is running in the logical dump. Suppress all system information. This is equivalent to: DPEDIT -MEM -NO_SYS -NP -GP my_group-id

-NS

Do not dump the sharable or globally sharable bound units in the logical dump.

-NO_SYS

Do not dump the system area in the logical dump.

-PSYS

Limit the physical dump to the system area.

-FORCE

If the error "DUMPFILE IS INCOMPLETE" (defined below) appears, this argument causes DPEDIT to ignore this condition and to try to process the file anyway. Note that since part of the memory image is missing, it may not be possible to get a logical dump.

NOTE

If no arguments are specified, the default is to do a logical and physical dump of memory.

Example 1:

```
DPEDIT ^DMPVOL>DUMPFILE -NL -TO X'3000'
```

This command loads the Dump Edit utility and requests only a physical dump of the first 12K locations of the specified dump file.

Example 2:

```
DPEDIT -MEM
```

This command loads the Dump Edit utility and requests a logical and physical dump of current main memory.

Example 3:

```
DPEDIT -MEM -GROUP $S $D -NP -NF
```

This command loads the Dump Edit utility and requests a logical dump of only the System and Debugger groups from current main storage. The command suppresses display of the file management structures.

Example 4:

```
DPEDIT -MEM -GROUP XX -NP -NF
```

By specifying a group that does not exist (i.e., XX) this command requests an abbreviated logical dump consisting of only the System Summary of the currently executing system.

Operating Procedure for Dump Edit

The following steps must be performed before the Dump Edit program can be executed.

1. Mount the disk volume containing Dump Edit.

2. If Dump Edit is being used to print MDUMP output, mount the disk volume that contains the memory image obtained from the MDUMP memory dump.

3. Execute Dump Edit by specifying the DPEDIT command described previously.

DPEDIT processing can be stopped at any time by pressing the "BREAK" key. A **BREAK** message appears on the user's terminal display when processing stops. A GCOS 6 command may be specified at this point. If the Unwind (UW) command is specified, the end-of-processing details are automatically handled and control returns to the command processor with a successful subtask completion status. If the Start (SR) command is specified, DPEDIT resumes processing.

If DPEDIT appears to be looping, the loop can usually be broken and DPEDIT can be made to recover by forcing a **BREAK** and entering the Program Interrupt (PI) command. Note, however, that it is normal for DPEDIT to run for five or ten minutes while dumping a large memory or dump file.

DPEDIT Error Messages

Fatal errors terminate DPEDIT processing, return control to the command processor, and post an unsuccessful subtask completion status. Fatal errors include logical I/O errors and physical I/O errors as well as DPEDIT-specific errors. Fatal error messages are written to the error-out file. Error messages specific to DPEDIT are listed below. Additional information on error messages can be obtained in the Error Messages manual.

Immediately after execution of DPEDIT begins, and immediately before execution terminates, a message is written to the error-out file. These messages are explained in the description of the DPEDIT command.

Informational messages that generally reflect some condition peculiar to the data within the dump file may be interspersed with the dump information in the user-out file. These messages are provided to facilitate analysis of the dump and are listed below. A brief explanation of each message is provided. "^" in a message indicates that a parameter is supplied.

-MEM AND PATHNAME NOT ALLOWED ON SAME INVOCATION

Memory and dump file can not both be processed during a single invocation of DPEDIT.

ARGUMENT NOT RECOGNIZED

An invalid argument was given in the DPEDIT command line.

ATTEMPT TO INCREMENT A VIRTUAL ADDRESS BEYOND FFFFF

An internal error has occurred; the memory block dump routine has incremented beyond the largest virtual address.

DPEDIT CONTINUES AFTER A PI OR TRAP. P: ^ I: ^ LOAD ADR: ^

DPEDIT has trapped or a break, program interrupt has been executed. The P-register, I-register and load address at the time of the interruption are displayed and DPEDIT recovers.

DPEDIT MUST EXECUTE IN THIS POOL TO DUMP THIS STRUCTIRE FROM MEMORY

Because DPEDIT is executing in a different memory pool, it does not have visibility to the structure. Either execute DPEDIT from the current pool or take an MDUMP.

DUMPFILE IS INCOMPLETE

Either MDUMP did not complete properly or the dump file was too small to hold the complete memory image (see the - RCE argument).

DUMPFILE IS INCORRECT FILE TYPE

The dump file must be a non-UFAS relative file.

ILLEGAL NUMBER OF ARGUMENTS

Too many group names follow the -GROUP argument.

LAST VALID DUMP LOCATION REFERENCED: ^

Indicates the last valid dump address processed before an invalid dump address was found.

NEED MOD400 REL2.1 DPEDIT TO PROCESS THIS DUMPFILE

A release 3.0 version of DPEDIT has accessed a release 2.1 (or earlier) MDUMP file.

NULL BUD POINTER IN THE TCB

The pointer to the bound unit description in the task control block is null.

NULL LINK IN THE ^QUEUE

A null link was found in the specified hardware queue.

PHYSICAL ADDRESS IS NOT IN PHYSICAL MEMORY: ^

DPEDIT has encountered a physical address that is higher than the highest physical address of the system being dumped.

REQUIRED ARGUMENT MISSING

The address has not been specified for the -TO or -FROM argument.

THERE WERE ERRORS DURING THE EDIT

If the output of DPEDIT was directed to a file, errors that tend to appear frequently are only written on the file. If the errors occurred during the dump, this error is issued to the user's terminal.

THIS ADDRESS DOES NOT FALL WITHIN THE DUMP FILE: ^

The specified address is not within the scope of the dump file.

THIS BOUND UNIT WAS PREVIOUSLY DUMPED IN ^

The bound unit was previously dumped in the specified group or pool.

THIS SWAP POOL STRUCTURE CANNOT BE DUMPED FROM MEMORY

DPEDIT does not have visibility to the current structure. An MDUMP is required.

VIRTUAL ADDRESS EXCEEDS PHYSICAL MEMORY: ^

The specified virtual address represents a physical address that exceeds the highest physical address in the system being dumped.

VIRTUAL ADDRESS IS INVALID: ^

The specified virtual address exceeds FFFFF.

VIRTUAL ADDRESS NOT FOUND FOR ^. DUMP FILE IS SUGGESTED.

During a physical dump of memory, the specified physical address could not be translated into a valid virtual address for DPEDIT. An MDUMP is needed.

VIRTUAL ADDRESS OFFSET EXCEEDS SEGMENT SIZE: ^

The specified virtual address exceeds the segment size in the corresponding segment descriptor.

VIRTUAL ADDRESS REFERENCES INVALID SEGMENT: ^

The segment descriptor for this specified virtual address is invalid.

INTERPRETING AND USING MEMORY DUMPS

This subsection describes significant locations in memory dumps, how to interpret the contents of locations on memory dumps, and how to use memory dumps to perform the following procedures:

- Finding the location in memory of your code
- Determining where a trap occurred
- Determining the state of execution of your code.

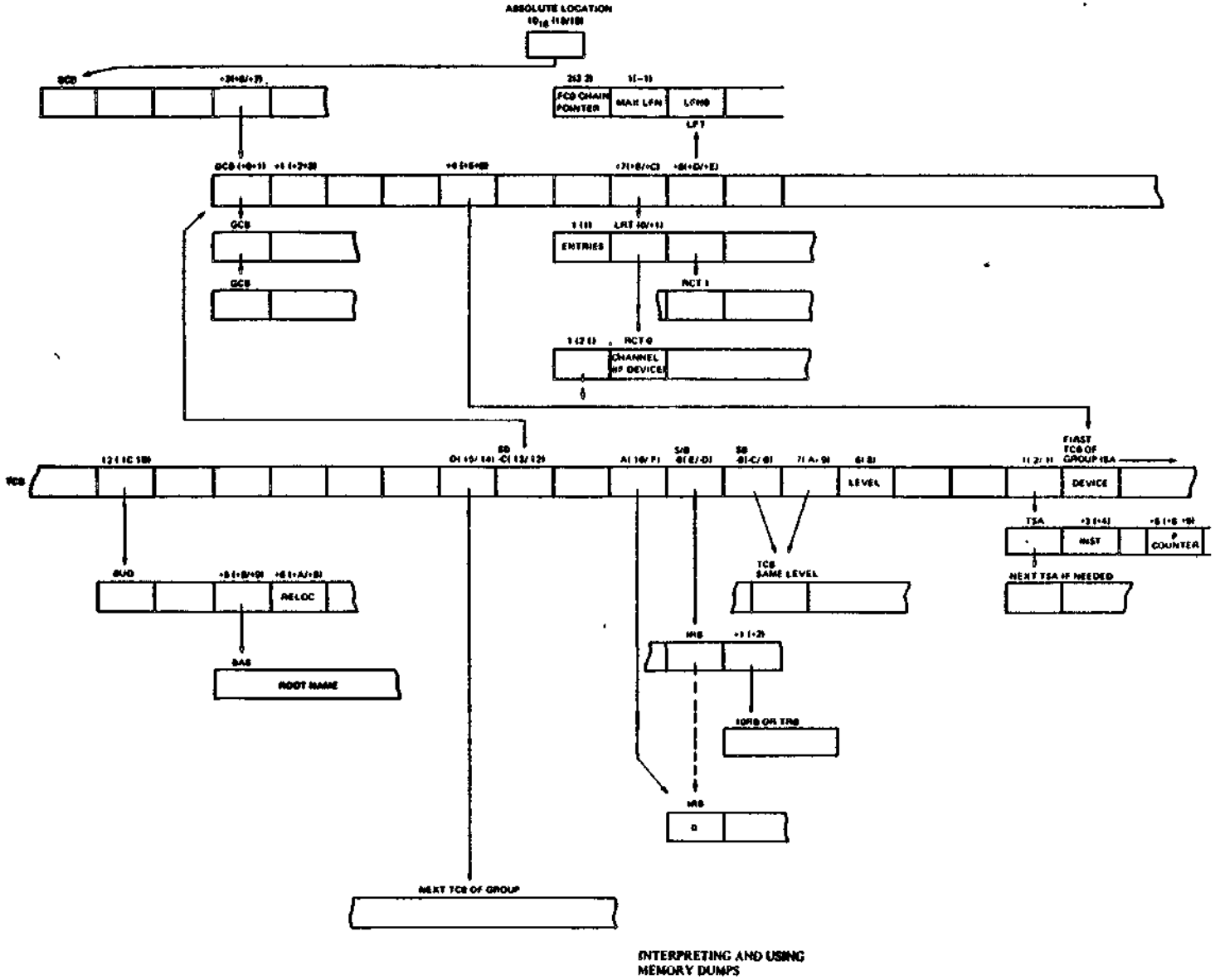
A trap is a special software- or hardware-related condition that may occur during the execution of a task. Many traps are caused by an error, but a few, such as the Monitor Call, are not. The above procedures may have to be performed if a trap message is issued. Traps are described in Appendix A.

SIGNIFICANT LOCATIONS ON MEMORY DUMPS

Table 19-2 describes memory locations on the dump that may be useful to refer to during debugging. It is assumed that you are familiar with the data structures referenced. Brief definitions of these data structures are contained in the glossary of the System Concepts manual. Figure 19-2 illustrates a map of systems data structures.

Table 19-2. Significant Locations on Memory Dump

Memory Address	Meaning
0010/0011	Head of queue of available trap save areas (TSAs).
0018/0019	Pointer to system control block (SCB). This is the key to locating all system data structures.
0020-0023	Level activity flags for levels 0 through 63. Bits ON indicate which levels are ready to execute; the lowest (numerically) of these levels is the level currently executing (i.e., the active level). The level 63 bit always is on. The clock level bit (4) may be on, and the debug level bit is on if the dump resulted from a Multiuser Debugger or a $D DEBUG DP directive.
0024-007F	Trap vectors. Each trap vector is associated with a specific trap condition and points to that trap handler's entry address. The trap vector for trap number 1 is in location 7E/7F. The trap vectors for subsequent trap numbers are in descending, contiguous locations; i.e., the trap vector for trap number 2 is in location 007C/007D.
0080-00FF	Pointers to interrupt save areas (ISAs) for levels 0 through 63, respectively. A null value means there is no dedicated task (i.e., driver) or nondedicated task ready to execute on the specified level.



ENTERPRETING AND USING MEMORY DUMPS

Figure 19-2. Data Structure Map

Locations Relative to the System Control Block or Group Control Block

SCB+6/7	Pointer to the group control block (GCB) queue.
GCB+0/1	Pointer to next GCB in linked list of GCBs.
GCB+2/3	Task group identification ($S is the system group; $B is the batch group). The system will convert your user identification to non-ASCII representation.
GCB+D/E	Pointer to LFN 0 of logical file table (LFT).
GCB+B/C	Pointer to LRN 0 of task group's logical resource table (LRT).
GCB+5/6	Pointer to first task control block (TCB) of the group.
LRT-1	Number of entries in the LRT.

LRT+0/1

Pointer to LRN 0's resource control table (RCT); the RCTs for subsequent LRNs are in contiguous, ascending locations (LRT+1 points to LRN 1's RCT). A null entry indicates that the associated LRN is not used.

NOTE

Within an RCT, location 0 is the channel number of the resource if it is an input/output device.

RCT-2/-1

Pointer to task control block (TCB) for that resource.

Locations Relative to the Task Control Block (TCB) Pointer of the Desired Priority Level

TCB-8

Hardware-assigned priority level of the task.

TCB-1C/-1B

Pointer to current bound unit BUD.

TCB-10/-F

Pointer to top of queue of requests for the task.

TCB-E/-D

Pointer to end of queue of requests for the task (e.g., I/O requests for a driver).

TCB-13/-12

Pointer to the group control block (GCB) for the group to which this task belongs.

TCB-D -15/-14

Pointer to next TCB in this group.

TCB-A/-9

Pointer to last TCB on this priority level.

TCB-C/-B

Link to other task control blocks (TCBs) of the same or different task groups assigned to the same level.

TCB-2/-1

Pointer to the queue of trap save areas (TSAs) for the task. (Trap save areas are described in detail in Appendix A.) If a TSA is present, the task is executing system code or a user trap; if no TSA is present, check the program counter in the interrupt save area (ISA) portion of the TCB to determine the task's progress.

TCB+0

Device word, including channel number and level number. This entry is null if the task does not drive a device.

TCB+1

Hardware interrupt save area.

INTERPRETING THE CONTENTS OF A DPEDIT LOGICAL DUMP

This subsection describes memory dump interpretation when the DPEDIT logical dump format is used.

Finding the Location in Memory of Your Code

Locate your group-id and the TCB for your bound unit (BU). The first six characters of the BU filename are printed beside each TCB of the group in a logical dump.

The address at TCB-1C/- is the address of the bound unit (BU) description. The load address of the bound unit is found at this address +A. Calculate relative zero of the BU by subtracting the relative start address on its link map from this address.

Determining the State of Execution of Your Code at the Time of the Dump

Dump analysis begins with gathering all relevant information: the dump itself, the console hard-copy (if any) of the activity of a particular group (or groups), copies of the CLM_USER and >START_UP.EC files, plus any link maps.

These materials are required to understand the environment of the system represented in the dump.

Three conditions are discussed below:

1. Halt at level 2.
2. User level active at the time of dump.
3. No level active at the time of dump, except level 63.

HALT AT LEVEL 2

Examination of the level activity indicators at locations 20-23 confirms that level 2 is active. The system will force this condition to occur if either TSA or IRB resources are exhausted (see CLM SYS directive). Note that once level 2 becomes active, other lesser priority levels may activate but will not receive CPU time.

The D1 register contains an ASCII "IR" (4952) when IRB exhaustion has occurred. Location 10/11 is zero when TSA exhaustion has occurred.

If this symptom persists after augmenting the number of TSA/IRBs available to the system, it is possible that either your code or the system is improperly altering the TSA/IRB chains.

To verify this, take a memory dump immediately after system startup. This allows easy location of the TSA chains from location 10/11 and the IRB chains from the first location of the SCB. Compare this dump to one taken after all TSA/IRBs are supposedly exhausted to verify that they really are. If the system is suspect, supply both dumps to Honeywell. TSAs can also be exhausted by a recursive trap. A recursive trap uses up all available TSAs. Adding TSAs simply allows for greater recursion. In this instance, the system is suspect and dumps should be supplied to Honeywell.

The optionally configured defective-memory trap handler may also force a level 2 halt if a defective memory trap indicates the operating system's trap save area is exhausted. In this case, $R1 will contain X'DEFA'; $B1, the physical address of the defective memory; and $B2, the logical address of the defective memory.

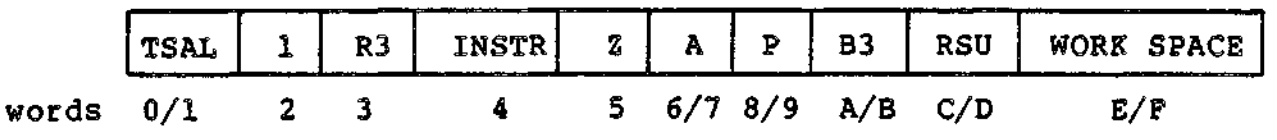
USER LEVEL ACTIVE AT THE TIME OF DUMP

This often indicates a halt or software loop condition on the active level. When a level is active, the pointer to the TCB associated with the code running is in the interrupt vector for that level. Match the TCB pointer with the TCBs listed for the groups present in the system. When a level is active, use the P-counter in the ISA portion of the TCB to locate the software running at the last time this level's context was saved. Since the system clock is active on level 4, the P-counter in the ISA for this level is usually helpful. It is also helpful to record the contents of R/B registers and E0 when entering STEP mode at the control panel prior to taking the dump.

NO LEVEL ACTIVE AT THE TIME OF DUMP

This condition usually indicates a system failure in that all tasks have been suspended and none are being reactivated. In this situation it is helpful to determine the conditions existing at this time. To do this, examine all TCBs in groups other than the $S group. If the TCB under examination has not experienced a default trap condition, it may or may not have an associated TSA. If a TSA is shown, DPEDIT will display the monitor call function code if the trapped instruction is 0001 (monitor call generic).

When the system is called for a monitor function, only those registers that must be preserved by the system are saved in the TSA workspace. The saved registers are: B7, B6, B5, B1, R5, R4, M1, beginning at TSA location E/F. The trap save area (TSA) is illustrated below:



Determining Where a Trap Processed by the System Default Handler Occurred in Your Code

If a trap message occurs on the operator terminal from the system default trap handler, i.e., (id) BUname (0303zz) level, the TCB of the referenced task group may be located using the bound unit name (BUname). In this situation, unless the TCB is subsequently requested, the last two areas associated with the TCB are related to the system handling of the trap. The first TSA following the TCB was used by the system to forcibly terminate the task request in progress when the trap occurred. Your information is found in the next TSA associated with the TCB. It contains the hardware information described in the previous section of this appendix, followed by a complete set of registers current when the trap occurred. The order of the registers, beginning at location E/F of the TSA, is: B7, B6, B5, B4, B2, B1, I, R7, R6, R5, R4, R2, R1, M1 (B3, R3, I are already in the TSA). When the TCB has been rerequested, only this second TSA remains attached to the TCB.

FINDING THE LOCATION IN MEMORY OF YOUR CODE

The three activities above may be performed from the DPEDIT physical dump presentation. The examination of TCB contents is the same once the TCB is located. Use the following procedure to find the TCBs for your group.

1. Go to location 0018/19; this location contains a pointer to the system control block (SCB).

2. Go to location SCB+6/7; this location contains a pointer to the group control block (GCB) queue; GCB+0/1 links to the next GCB in the queue. Determine the group id at GCB+2/3 is your group id.

3. Go to location GCB++5/+6 to determine the location of the task control block (TCB) queue of the task group.

4. Go to location TCB-1C/-1B to determine the location of your current bound unit descriptor (BUD).

5. Go to location BUD+A/B. This location is the relocation factor of the bound unit; your code should start at this location.

6. Go to location BUD+8/9; this location points to the location of the bound unit attribute section (BAS).

7. Go to location BAS+0 to determine the bound unit's root name; this name should be the same as the bound unit's file name.

8. If you did not find the root name for which you were looking, go to location TCB-15/-14; this location points to the next TCB of the task group. Follow through the chain of TCBs until you find your task's task control block.

PRINTING AN INCOMPLETE MEMORY DUMP

By specifying the DPEDIT command with the -FORCE argument, an incomplete memory dump may be printed. See the DPEDIT command definition earlier for information on requesting the incomplete memory dump.

REQUESTING AND PRINTING MLCP DUMPS

The Dump Communications Processor (DCP) utility is a system maintenance tool that allows you to obtain a computer printout of the formatted contents of the Multiline Communications Processor (MLCP). The following MLCP Random Access Memory (RAM) areas are displayed for each channel connected to the processor:

- Line Control Table (LCT). A system structure that contains control information for the line channels. Each LCT is 64 bytes long: the first 32 bytes are for the receive channel; the second 32 bytes are for the transmit channel.

- Channel Control Blocks (CCBs). Data structures that are used by the MLCP firmware to control the flow of data between a main memory program and the MLCP. Each CCB is eight bytes long.

- Channel Control Programs (CCPs). Microcoded programs that process data characters, protocol headers, and framing characters. CCP functions are data character editing, communications pac control, parity and cyclic redundancy checks, and error detection and handling.

The dump of the communications processor is destructive in that the context of the channel used for communication between the Central Processing Unit (CPU) and the MLCP is lost until the system is rebootstrapped. When you are trying to pinpoint a problem with a particular channel, do not use that channel to dump the communications processor because the information on the printout for that channel would only apply to the dump operation itself. The dump channel must be in the same MLCP as the information to be collected.

MEMORY POOL CONFIGURATION REQUIREMENT

To view areas of the MLCP RAM using the DCP utility, the memory pool of the task group in which DCP runs must be configured as privileged, unprotected, and non-contained at system startup. To configure the memory pool at system startup, use the Configuration Load Manager (CLM) MEMPOOL directive: do not specify the U (Unprivileged) and P (Protected) values; do specify the NC (Non-Contained) value. See the System Building and Administration manual for the full description of the MEMPOOL directive.

DCP COMMAND

Write to the-user out file (the line printer attached to the system) the formatted contents of the Multiline Communications Processor.

FORMAT:

```
DCP[?SILENT] [ctl_arg]
```

NOTE

[?SILENT] is an optional entry point that suppresses the welcome message.

ARGUMENTS:

-DC [dump_channel]

The optional hexadecimal channel number to be used destructively to dump the communications processor to the Central Processing Unit (which then sends the information to the user-out file). dump_channel must be in the same MLCP as the information to be collected.

Default: channel 0000

NOTE

Channel 0000 is an invalid MLCP dump_channel. Thus, failure to specify a dump_channel will result in an unavailable resource trap.

-HMA [hi_mem_addr]

The optional hexadecimal High Memory Address (HMA) at which the dump of the CCP area terminates.

Default: Dump the entire CCP area.

-RHU

Dump all areas designated as Reserved for Hardware Use (RHU).

Default: Do not dump RHU areas.

DESCRIPTION:

The DCP command writes to the user-out file the formatted contents of the communications processor associated with dump_channel. For each channel of the MLCP, the following are displayed:

- Line Control Table (LCT)
- Channel Control Blocks (CCBs)
- Channel Control Program (CCP) area, up to and including the specified High Memory Address (HMA)
- Reserved for Hardware Use (RHU) areas, if required.

DCP is self-documenting. Enter

```
DCP ?
```

and the DCP command line format is displayed.

DCP can be run as an operator command in the system group or as a user command in any other group that has execution privilege.

The information on the printout concerning the channel selected as the dump_channel applies only to the dump operation itself.

A hexadecimal number is represented by the appropriate sequence of hexadecimal digits optionally preceded by an X.

The following error message is specific to DCP:

```
172C  INVALID ARGUMENT nn
```

SAMPLE DCP PRINTOUT WITH COMMENTARY

Figure 19-3 shows a DCP printout of LCTs, CCBs, and CCPs contained in MLCP RAM for the indicated channels and byte addresses. The command that obtained the printout is:

```
DCP -DC F380
```

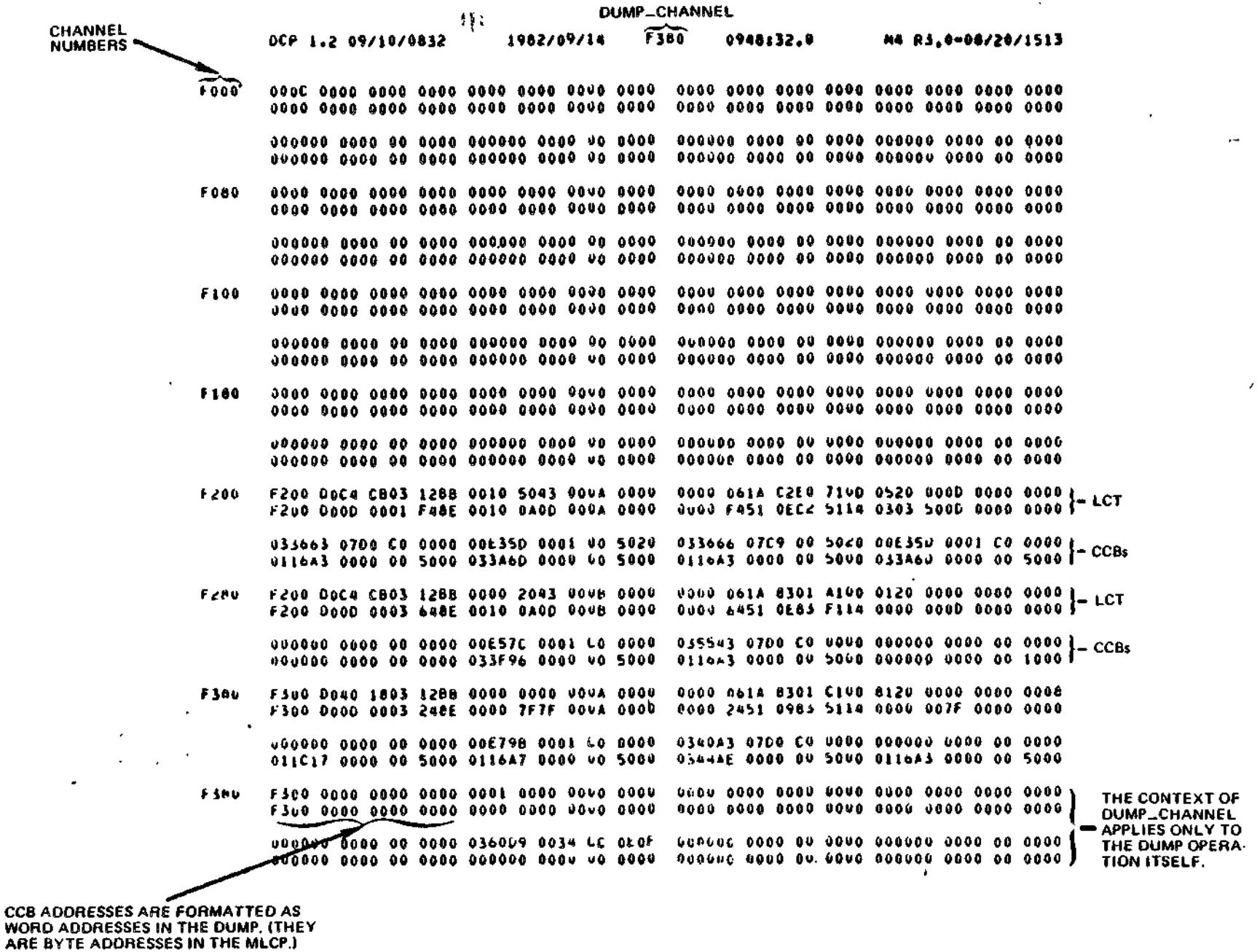


Figure 19-3. Sample DCP Printout

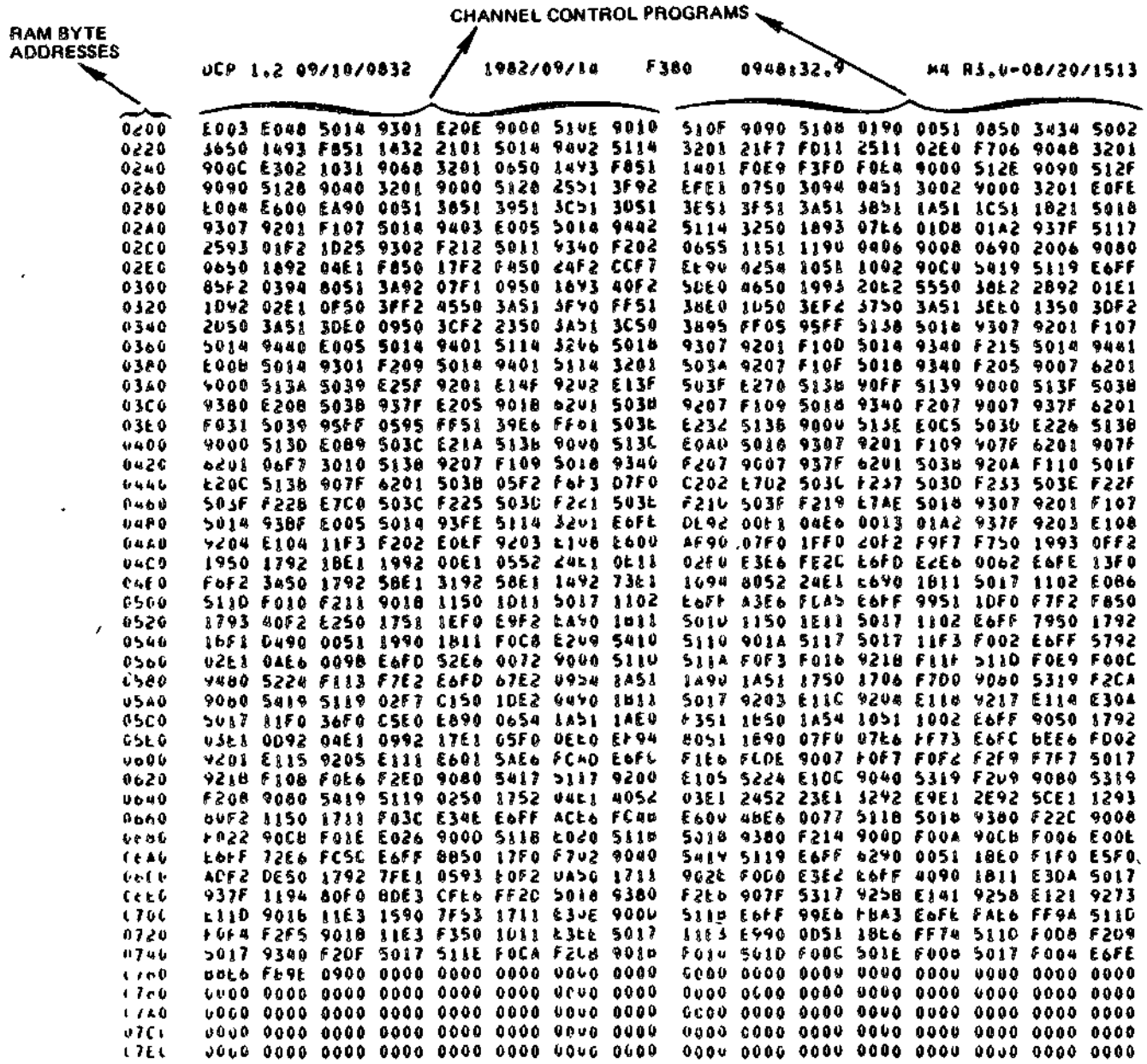


Figure 19-3 (cont). Sample DCP Printout

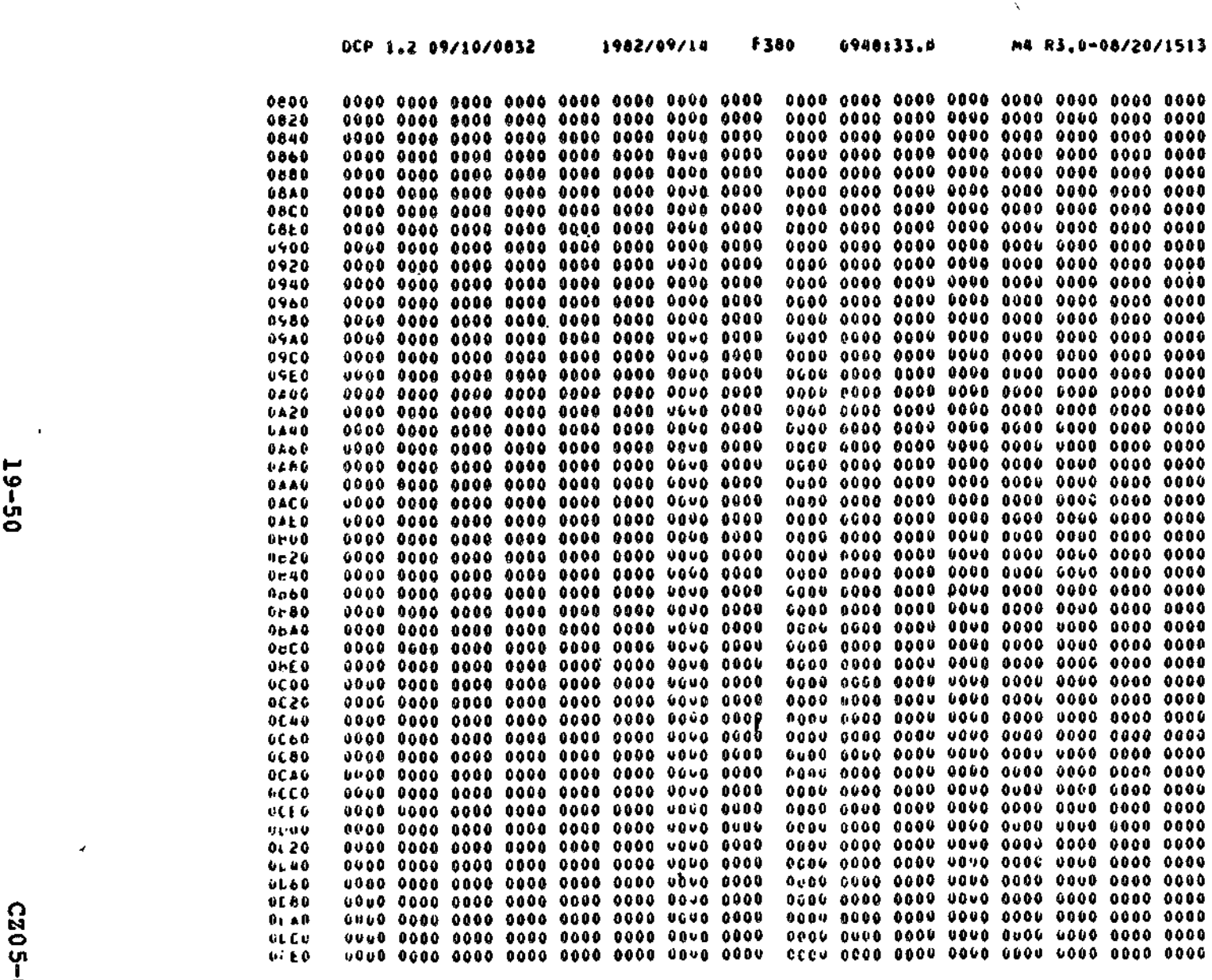


Figure 19-3 (cont). Sample DCP Printout

7. Patch

Section 20
PATCH UTILITY

The Patch utility is used to apply patches to and remove patches from object units and bound units. Patches are identified by patch-ids. The Patch utility can also be used to list, by patch-id, patches already applied to an object unit or bound unit. The listing is written to the user-out file, terminal line screen, or printer for a hard copy.

The Patch utility, in modifying object or bound units, will extend the file space, as necessary. Insufficient file space will terminate Patch operations; therefore, you should ensure that sufficient space exists to accommodate the patch(es) on the medium (disk, etc.).

USING THE PATCH UTILITY

Patch execution is controlled by directives entered to Patch through the operator's terminal, user terminal, a card reader, or a sequential file. The Patch utility operates in batch mode or in interactive mode. Each mode is described separately below.

Batch Mode

In batch mode the user can:

- Modify a bound unit's shared and system attributes by setting/clearing bits in the bound unit's attribute table
- Assign a patch revision number to a bound unit

- Assign an address to an undefined external reference
- Interrogate the current contents of a bound unit
- Apply a patch with or without verifying the existing value of the location to be patched
- List patches
- Eliminate patches.

Patch processes as they are entered directives that modify a bound unit's attribute or version number and interrogate the bound unit. Regardless of the input sequence of other directives, Patch processes them in the following order:

1. Eliminates patches
2. Defines undefined external references
3. Applies patches
4. Lists patches.

Interactive Mode

By specifying the Patch command with the -IA argument, a bound unit can be patched in interactive mode. In interactive mode, Patch directives must be completed before they are applied; a directive is completed when the Patch utility reads a new directive. Only the file specified on the Patch command line can be patched with each invocation of the Patch utility.

Version number processing, manipulation of the shared or system attributes, and interrogation are always performed as the directives are keyed-in.

The Patch directives are listed and briefly defined below. Detailed descriptions for each Patch directive are provided later in this section.

Directive	Directive Name
CLSY	Clear system bit
DP	Apply patch(es) to data section of bound unit or to common area of object file
EP	Eliminate named patch or all patches
GO	Process previous patch directive
GNSH	Set global share bit off
GSHR	Set global and root share bits on

Directive Name	Function
HP	Apply hexadecimal patch(es) to specified file
LDEF	Assign an address to an undefined external location reference
LN	List patches but do not exit from Patch
LP	List patches and exit from Patch if mode is batch
LS	List patches by name only and exit from Patch if mode is batch
NS	Set share bit off
Q	Process previous patch directives and exit from Patch
SD	Apply symbolic data patch(es)
SP	Apply symbolic patch(es)
SS	Set share bit on
STSY	Set system bit on
VDEF	Assign a value to an undefined external symbol
VN	Verify or change revision number of bound unit
WA	Interrogate bound unit
*	List a comment on the user-out file

LOADING PATCH

To load Patch, enter the PATCH command, as follows:

FORMAT:

```
PATCH filenm [ctl_arg]
```

ARGUMENTS:

filenm

Pathname of the object unit file or bound unit file to be patched. If an object unit is being patched, the last two characters of the pathname must be .O.

ctl_arg

The following control arguments may be entered:

-IA

Operate in interactive mode. Process one directive at a time; error messages (if any) immediately follow the applicable directive. If this argument is not specified, Patch operates in the batch mode.

-IN path

Pathname of the device through which Patch directives will be entered; can be the operator terminal, another terminal, a card reader, or a sequential file. Error messages are written to the error-out file. Patch error messages are described in the System Messages manual.

Default: The task group's current user-in file.

-M6

Bound unit to be processed was created by the MOD 600 Linker.

This argument should not be used when patching an object file. (The .O at the end of the filenm on the command line identifies to Patch that the file is an object file.)

Default: MOD 400.

{-PROMPT / -PT}

If input is from the operator terminal or another terminal, each time the PATCH utility program is ready to accept an input line, the typeout P? appears on the input device.

Default: No prompt.

-SI

Suppress the display of the sign-on message (i.e., PATCH, followed by the revision number and the date patch was created).

Default: Patch sign-on message is displayed.

{-SIZE n / -SZ}

Create a patch work area of n 1024-word blocks of memory.

Default for n: 1.

SUBMITTING PATCH DIRECTIVES

Each Patch directive consists of only a directive name or a directive name followed by one or more values. Values must be separated by a delimiter. The delimiter can be a space, a comma, or a semicolon. However, on an interactive device (i.e., a terminal), the carriage return replaces the delimiter. Lines may neither begin nor end with a comma or semicolon. If directives are entered from a card reader, trailing blanks or column 80 replace the delimiter.

Multiple Patch directives may be specified during one execution of the Patch utility. To patch another bound unit or object unit, Patch must be re-executed.

For patching in the interactive mode:

- Patch directives are processed in the sequence in which they are entered.
- Patch directives can be entered in any order, except that Quit (Q) must be entered last.
- A patch directive must be complete before it is processed; it is complete when Patch reads a new directive.

For patching in the batch mode:

- The List Patches Now (LN) directive must be the first directive; otherwise, it is processed like an LP directive.
- Patches are first eliminated, then applied, and finally listed regardless of the sequence in which the associated directives are entered.
- The version number directives (VN), the share bit and systems bit directives (SS, STSY, CLSY, GSHR, GNSH, and NS) are always processed interactively in the order in which they are entered.
- The WA directive is processed when it is entered.

If directives are being entered through the operator terminal or another terminal, press RETURN at the end of each line. Each time RETURN is pressed, except after quit, the typeout P? is reissued if the prompt control argument was specified in the command line.

To enter Patch directives for a different file, you must reload Patch, specifying a different file in the filenm argument.

PATCHING TECHNIQUES

Techniques used when "Naming the Patch" and "Applying the Patch" are described in the following paragraphs.

Naming the Patch

Each patch has a patch-id by which it is identified. When applying patches with the DP, HP, SD, or SP directives, you must specify a patch-id. The patch-id identifies the patch(es) and specifies whether the patch(es) are to be applied to an object unit, root, or overlay of a bound unit. To eliminate patches from an object unit or bound unit, you must specify in the Eliminate Patch directive the patch-id with which the patch(es) are associated. See "Data Patch Directive (DP)" for a description on how to designate patch-ids.

Applying the Patch

If an object unit is being patched, object records are created for the specified patches and appended to the end of the object file. When the object unit is processed by the Linker, existing values are replaced with the specified patch values. Locations that contain external references should not be patched; results are unspecified.

If a bound unit is being patched, each specified patch value is applied directly to the proper image record in the bound unit. The previous value, the patch-id, and the patch value are saved in a Patch history record that is written at the end of the file area allocated to the bound unit. This record is referred to each time a List Patch or Eliminate Patch directive is specified.

NOTE

Use caution when patching executing bound units. If a program or one of its overlays is loaded while in the process of being patched, results are unspecified.

PATCH DIRECTIVES

The Patch directives are described on the following pages in alphabetic order by directive name.

CLEAR SYSTEM BIT

Turn off the system bit in the bound unit's attribute table. This directive prohibits the patched bound unit from executing in the system ($S) group. The CLSY directive is not allowed for object files.

FORMAT:

```
CLSY
```

NOTE

The system bit was initially set at link time by the SYS Linker directive.

COMMENT

COMMENT

List the accompanying text on the user-out file. The contents of the Comment directive are not saved.

FORMAT:

```
* comment-text
```

DATA PATCH

Apply (for bound units) one or more hexadecimal patches, by relative location, to the data section of the bound unit. The bound unit must have separate code and data sections, and have been created by the Linker when the -R Linker ECL argument is specified.

For object files, the DP directive causes patches to be applied to common areas.

FORMAT:

For Bound and Load Units, Without Verification:

```
DP patch-id /addr patchval[ patchval...][ /addr
    patchval...]
```

For Bound and Load Units, With Verification:

```
DP patch-id /addr (verval patchval[ verval patchval...])
    [/addr (verval patchval[ verval patchval...])]
```

For Object Files, Without Verification -- Local Common Block:

```
DP patch-id /offsetl patchval[ /offsetl patchval]...
```

For Object Files, With Verification -- Local Common Block:

```
DP patch-id /offsetl (verval patchval)[ /offsetl
    (verval patchval)]...
```

For Object Files, Without Verification -- Named Common Block -- One Blockname Per Directive:

```
DP patch-id blockname /offset patchval[ patchval...]/
    offset patchval [patchval... ...]
```

For Object Files, With Verification -- Named Common Block -- One Blockname Per Directive:

```
DP patch-id blockname /offset (verval patchval[ verval
    patchval]...)[/offset (verval patchval[verval
    patchval]... ...)]
```

ARGUMENTS:

patch-id

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces; the last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number; the first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

/addr

Relative location at which the first (or only) subsequent patch value will be applied. Each address must comprise one to six right-justified, hexadecimal characters, and must be preceded by the slash character (/). Subsequent patch values, if any, are applied to succeeding memory locations.

NOTE

Care must be taken in specifying an address to be patched. If the address of a location to be patched is identified when a bound unit is being executed, that memory address contains three possible factors:

1. The original address of the location in the bound unit relative to the beginning of the bound unit.

2. The linking relocation factor.

3. The loader relocation factor.

If the address is identified at execution time and the bound unit is to be patched, the loader relocation factor must be subtracted from the address

identified in the executing bound unit. If the object unit is to be patched, both the linking and loader relocation factors must be subtracted. Object unit locations can also be obtained through examination of the listing produced during assembly of the object member.

offsetl

Non-negative offset from the beginning of $LCOMW.

patchval

A value of one to six hexadecimal characters to insert into $LCOMW. Relocatable values are not permitted and only one patch value can be specified for each patch.

blockname

Symbolic name of the common block. The name can contain one to six characters.

offset

Offset from the symbol name of the common block.

/patchval

Value to be inserted at an address, replacing the contents of that location. The value must be specified as one of the following:

1. Data, represented by one to four hexadecimal characters.

2. Relocatable address, represented by one to six hexadecimal characters, preceded by the character <.

verval

Verification value; one to six hexadecimal characters specifying value that should be in location before patch is applied.

NOTES

1. Each verval must be immediately followed by a patchval.

2. The verification value(s) and patch value(s) associated with each address must be enclosed within parentheses.

3. For consecutive locations, the old and new values can be included within one set of parentheses. The /addr field is adjusted internally by Patch.

4. Within a set of parentheses, the number of old values must equal the number of new values.

5. The IMA indicator cannot be used with an old value. IMA status is determined by Patch from the module or from the new value.

6. For SLIC or LAF IMAs, old value and new value can be up to six characters.

7. For SLIC or LAF IMAs, Patch allocates two words. For example, assume that the following directive applies to a SLIC module:

   DP patch-id,/100,(1111,<12345,ABC,DEF)

   If the contents of 100 and 101 are 001111, and the contents of 102 are ABC, the patch will be applied, and as a result the contents of the specified addresses will be:

Address	Contents
100	01
101	2345
102	DEF

8. Verified and nonverified patches can be included within one patch directive; however, if the verify fails, none of the addresses in the directive are patched.

9. A left parenthesis cannot immediately follow a right parenthesis. There must be a /addr field between them.

10. In a bound unit, an IMA may be patched to a non-IMA or a non-IMA patched to an IMA.

11. In object modules, patches to areas that have no defined value cannot be verified.

12. In a bound unit, if the new value is not an IMA, the old value can be no more than four hexadecimal characters even if the old value is an IMA.

13. SLIC means SAF/LAF independent code. MOD 400 bound units are usually LAF, but SLIC bound units may still be patched and executed.

ELIMINATE PATCH

ELIMINATE PATCH

Eliminate all patches associated with a specified patch-id from the designated object unit or bound unit. The patch(es) must have been previously applied by DP, HP, SD, or SP directives. To determine what patches have been applied, and their patch-ids, enter one of the list patch (LN, LP, LS) directives described later in this section.

FORMAT:

```
EP  patchid
    ALL
```

ARGUMENTS:

patchid

Patch-id of the patch(es) to be removed. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces; the last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number, the first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

ALL

If the ALL option is used, all patches in the file are eliminated.

GO

Tell Patch that the previous directive is complete and is to be processed. This directive is effective only in the interactive mode. In the interactive mode, a new Patch directive signals the end of the previous one. The GO directive is used in circumstances in which the user would like to have a directive processed before entering any other directive.

FORMAT:

```
GO
```

HEXADECIMAL PATCH

Apply one or more individual patches, by relative location, to an object unit or bound unit.

If a bound unit is being patched, you can designate that specified patch(es) be applied only if specified location(s) currently contain specified value(s); these are called verification values. Within a single HP directive, verification values may be specified for some or all of the locations. If any of the verification values do not match the values currently at the locations for which verification values were specified, none of the patches specified in the HP directive are applied.

FORMAT:

Without Verification Values:

```
HP patch-id,[base,]/addr,patchval[,patchval...patchval]
    [,/addr,patchval[,patchval...patchval]]...
```

With Verification Values:

```
HP patch-id,[base,]/addr,(verval,patchval[,verval,
    patchval])[,/addr,(verval,patchval[verval,
    patchval])]
```

NOTES

1. One or more lines of arguments may be specified. When two or more lines of arguments are entered for an HP directive, the last character on each line must be a valid hexadecimal character or right parenthesis. Individual fields, values, and addresses must not be split between lines. The entry of a Patch directive name (e.g., EP, LP) at the beginning of a line designates the end of the previous Patch directive.

2. A space may be used in lieu of a comma as a separator.

ARGUMENTS:

patchid

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces; the last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number; the first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

base

Optional argument allowed only for bound units. Base defines a value that is added to all locations; i.e., /addr specified in the associated DP, HP, SD, or SP directives and all IMA references. If this argument is omitted, the default value is zero. Base can be entered as a hexadecimal address of one to six characters or as a name that has been specified as an EDEF at link time and placed in the bound unit symbol table. If a symbol name is used, Patch finds the name in the symbol table and uses its address as the base value. The format for the symbol name as a base is +symname, where symname comprises 1 to 12 characters. If a hexadecimal address is used for base, the plus sign is not required.

/addr

Relative location at which the first (or only) subsequent patch value will be applied. Each address must comprise one to six right-justified, hexadecimal characters, and must be preceded by the character /. Subsequent patch values, if any, are applied to succeeding memory locations.

NOTE

Care must be taken in specifying an address to be patched in either an object unit or a bound unit. If the address of a location to be patched is identified when a bound unit is being executed, that memory address contains three possible factors:

1. The original address of the location in the object unit relative to the beginning of the object unit

2. The linking relocation factor

3. The loader relocation factor.

If the address is identified at execution time and the bound unit is to be patched, the loader relocation factor must be subtracted from the address identified in the executing bound unit. If the object unit is to be patched, both the linking and loader relocation factors must be subtracted. Object unit locations can also be obtained through examination of the listing produced during assembly of the object unit.

patchval

The value to be inserted at an address, replacing the contents of that location. The value must be specified as one of the following:

1. Data, represented by one to six hexadecimal characters

2. Relocatable address, represented by one to six hexadecimal characters, preceded by the character <.

verval

Verification value; one to four hexadecimal characters specifying value that currently should be in location at which subsequent patch will be applied. See the notes on verification that follow the DP directive.

Example 1:

```
HP PTCHIDRT,/1B2A,1FFF,1DFC,<2BFC,2D4E,<ABF2
```

This Hexadecimal Patch (HP) directive requests that the subsequent patches, identified by the name PTCHIDRT, be applied to the root. Patch values $1FFF_{16}$ through $<ABF2_{16}$ are to be inserted in successive locations, with the first patch value $1FFF_{16}$ to be located at address $1B2A_{16}$. The hexadecimal patches are to replace any previous values in these locations. The value to be inserted in address $1B2C_{16}$ is the two word addresss $2BFC_{16}$, which is to be relocated at load time; the relocatable address $ABF2_{16}$ is to be inserted in address $1B2F_{16}$.

Example 2:

```
HP VPATCH01,/1FEA,(1A1,1B7,1A7,1B8),/1E72,8900
```

This example illustrates the use of verification values in a Hexadecimal Patch (HP) directive requesting that specified patches, identified by the name VPATCH01, be applied to overlay 01. Patch will check location $1FEA_{16}$ for the value $1A1_{16}$, and location 1FEB for the value $1A7_{16}$; if the values are at those locations, then the contents of locations are changed as follows: location $1FEA_{16}$ will contain $1B7_{16}$. location $1FEB_{16}$ will contain $1B8_{16}$, and location $1E72_{16}$ will contain 8900_{16} . If either of the verification values is incorrect, none of the three locations will be changed.

INTERROGATE BOUND UNIT

INTERROGATE BOUND UNIT

Display on the user-out file the current contents of locations specified by this directive. This directive cannot be used to display locations in object files.

FORMAT:

WA,[ovly,]/addr$_1$ [,words][,/addr$_2$...]

ARGUMENTS:

ovly

Overlay number in hex that the address references. If this field is omitted, the root is the default. The root can also be specified as RT. For -R type bound units, this field can be DP for data section or RT for code section as well as being an overlay number.

addr

Specify the hex address within specified root or overlay indicating where the display is to start.

words

Number of consecutive words to display. The default is 1.

LDEF

Assign a specified address to an undefined external location reference and change all locations that reference this name. This directive is not allowed for object files.

FORMAT:

LDEF;symname;[<]addr[;L]

ARGUMENTS:

symname

Name of the undefined external reference that will be assigned an address; can be from 1 to 12 characters in length.

addr

Address to which symname will be assigned.

[<]

Address specified is an IMA address. If this argument is not specified, the address is treated as P+DSP.

[;L]

List all changed external references to symname on the device specified as user-out.

Default: No list.

Undefined external references in a bound unit can only be changed one time. If you make a mistake, you must use HP patch directives to correct each location containing the wrong information.

NOTE

The user should be aware that there is no history kept of the changes that are made when the LDEF directive is used. It is wise, therefore, to utilize the L argument and retain the listing for future reference.

Example 1:

```
LDEF;EPPTR;50;L
```

This directive assigns address 50 to symbol EPPTR and lists all locations that are changed to reference the address 50.

Example 2:

```
LDEF;PK;<50;L
```

This directive assigns address 50 to symbol PK and changes all IMA references to external symbol PK to address 50.

LIST PATCHES

Produce a listing of all patches within the object unit or bound unit being patched. The listing is produced on the user-out file.

If a bound unit is being patched, the listing designates, for each patch, the following information in the order listed: full patch-id, address at which the patch was applied, contents of the location before the patch was applied, and the patch value.

NOTES

1. In the listing, the characters that identify the root or overlay appear first, and are separated from the other character constituting the id by spaces. When a bound unit is being patched in a common area, the letters CM are printed rather than RT.

2. If termination of the listing of patches is desired before normal completion of the list process, use the BREAK facility followed by a NEW_PROC command. The PATCH program must then be reloaded.

FORMAT:

```
LP
```

Example:

```
0001 NOHLT3 000002E2 00000000 00000F02
```

This printout is one line of a listing of patches applied to a bound unit being patched. The printout has the following meaning: a patch identified by the patch-id NOHLT3 was applied to overlay 01. The patch was applied to location 02E2; this location previously contained 0000, and now contains 0F02.

If an object unit is being patched, the listing designates, for each patch, the following information in the order listed: patch-id (excluding the last two characters, which identify the root), address at which the patch was applied, and the patch value.

Example:

```
NUMBRF  00000162   00000444
        00000163   00000222
NUMBRH  000001A6   00000333
        000001A7   00000444
        000001A8  <00000221
        000001AA   00000004
        000001AC  <00000321
```

This typeout is a listing of patches applied to an object unit being patched. The first line designates that patch 0444, whose patch-id is NUMBRF, was applied to location 0162. Note that the last two characters of the patch-id (i.e., RT) were omitted from the printout.

LIST PATCHES NOW

List all patches in the specified file and then allow more patches to be applied. This directive is effective only in batch mode and can be applied only to bound unit files. It must be the first directive issued. If it is not the first directive, or if it is entered in interactive mode, it is processed the same as an LP directive. The LN directive allows the current patches to be listed and additional patches to be applied without reloading Patch.

FORMAT:

LN

Example:

```
0000  CONRCT  000000A8  0005A4D  0005A4E
```

This printout is one line of a listing of patches applied to a bound unit being patched. The printout has the following meaning: a patch identified by the patch-id CONRCT was applied to overlay 00. The patch was applied to location 000000A8; this location previously contained 0005A4D, and now contains 0005A4E.

LIST PATCH NAMES

LIST PATCH NAMES

List the names (patch_ids) of the patches in the specified file. Addresses and values are not listed.

FORMAT:

LS

Example:

```
0000 CONRCT
```

The printout is one line of a listing of patches applied to a bound unit being patched. The printout has the following meaning: The patch identified by patch-id CONRCT was applied to overlay 00.

LIST SPECIFIED PATCH

List those patch ids specified. Up to five patch ids can be requested per run.

FORMAT:

```
LS patchid [;PATCH_id...]
```

Example:

```
LS NUMBRART; NUMBRB00
```

In this example, the directive will cause the entire patch NUMBRART and the entire patch NUMBRB00 to be listed.

QUIT

QUIT

Inform Patch that the last Patch directive has been entered, and initiate processing of the specified Patch directives. This directive should be preceded by at least one other Patch directive. When the directive(s) have been executed, execution of Patch terminates.

FORMAT:

```
Q
```

SET GLOBAL SHARE BIT OFF

Turn off the global share bit in the MOD 400 bound unit. The share bit of the root is not affected by this directive. This directive cannot be used in MOD 600 systems nor object unit files.

FORMAT:

```
GNSH
```

SET GLOBAL SHARE BIT ON

SET GLOBAL SHARE BIT ON

Set the global share bit of the root on in the bound unit. This directive cannot be used for MOD 600 bound unit or object files.

FORMAT:

GSHR

SET SHARE BIT OFF

Turn off the share bit of the root segment of a bound unit. Patch alters the status of the share bit only; it makes no check on the sharability of the module. This directive is not allowed for object files.

FORMAT:

```
NS
```

NOTE

This is the bit that is set on by the Linker directive SHARE.

SET SHARE BIT ON

SET SHARE BIT ON

Turn on the share bit of the root segment of a bound unit. Patch alters the status of the share bit only; it makes no check on the sharability of the module. This directive is not allowed for object files.

FORMAT:

SS

NOTE

This bit designates that the bound unit is sharable within a memory pool.

SET SYSTEM BIT ON

Turn on the system bit in the bound unit's attribute table. This directive must be employed if the patched bound unit is to execute in the system ($S) group. The STSY directive is not allowed for object files.

FORMAT:

```
STSY
```

NOTE

Before using this directive, consult with the person responsible for system building and determine the available memory. This Patch directive is equivalent to the Linker SYS directive.

SYMBOLIC DATA PATCH

Apply patches for object and bound units created by the Linker. For bound units, the directive causes patches to be applied to the data portion of separated object units. For object units, the directive causes one or more Assembly language one-word symbolic instructions to be applied to common areas, i.e., to either named or local common blocks. You can verify the current contents of locations while patching.

FORMAT:

For Bound Units -- No Verification:

SD patch-id/off$_1$ patchval$_1$ [/off$_2$ patchval$_2$...]

For Bound Units -- With Verification:

SD patch-id/off$_1$ (oldval$_1$;newval$_1$)[/off$_2$ (oldval$_2$; newval$_2$)...]

For Object Units -- Named Common Block -- No Verification:

SD patch-id;blockname;/offs;patchval$_1$ [patchval$_2$ ... patchval$_n$]

For Object Units -- Named Common Block -- With Verification:

SD patch-id;blockname;/offs;(oldval ;newval)[(oldval ;newval)...]

For Object Units -- Local Common Block -- No Verification:

SD patch-id;/offs;patchval

For Object Units -- Local Common Block -- With Verification:

SD patch-id;/offs;(oldval;newval)

NOTE

You can mix verification and nonverification patches. For example: SD NUMBRART;/135;(111;CMV $R7,8;2;STR $R6,=$R1);/150;ADD $R4,1000. Only the patches at locations 135 and 136 are verified.

ARGUMENTS:

patchid

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces; the last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number; the first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

offn

Non-negative offset from the beginning of the block.

oldval

Current contents of specified location. If the current contents are not oldvaln, all patches associated with patchid are not applied.

patchval (object units -- local common block)

Value to be inserted into the block. Relocatable values ae not permitted, and only one patch value can be specified for each patch address.

patchval (object units -- named common block)

Value to be inserted at an address, replacing the contents of that location. The value must be specified as

opcode field$_1$ [,field$_2$] [,field$_3$]

where opcode specifies an Assembly language instruction (except for I/O or floating point instructions); fieldn specifies either a register or a hexadecimal value.

blockname

Symbolic name of the common block. The name can contain one through six characters.

offs

Offset from the symbolic name of the common block.

patchval (bound units)

Value to be inserted at an address, replacing the contents of that location. The value must be specified as a symbolic instruction.

newval

Specify the patch value to be applied. See the appropriate description of patchval, above.

SYMBOLIC PATCH

Convert and apply one or more Assembly language symbolic instructions into the form of a hexadecimal patch. You can verify the current contents of the location while patching.

FORMAT:

Without Verification:

SP patch-id [;base] ;/addr$_1$;instruction$_1$
[;instruction$_2$...instruction$_n$]
[/addr$_2$; instruction/[instruction$_2$...instruction$_n$]]

With Verification:

SP patch-id [;base] ;/addr;(oldval$_1$;instruction$_1$
[;oldval$_2$;instruction$_2$...;oldval$_n$;instruction$_n$])

NOTES

1. One or more lines of arguments may be specified. When two or more lines of arguments are entered in an SP directive, instructions and verification values must not be split between lines. No line may begin with a semicolon (;). Individual fields, values, and addresses must not be split between lines. The entry of a patch directive name (e.g., EP, LP) at the beginning of a line designates the end of the previous patch directive. Hexadecimal patches are not permitted.

2. You can use a carriage return instead of a semicolon as a separator.

3. You can mix verification and nonverification patches. For example:

```
SP NUMBRDRT;/135;(111;LDV $R1,1;2;CL =
$R2);/150;STB $B2,400
```

Only the patches at locations 135 and 136 are verified.

ARGUMENTS:

patch-id

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: The first six characters can be any ASCII characters except spaces. The last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number. The first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

base

Optional argument allowed only for bound units. Base defines a value that is added to all locations; i.e., /addr specified in the associated DP, HP, SD, or SP directives and all IMA references. If this argument is omitted, the default value is zero. Base can be entered as a hexadecimal address of one to six characters or as a name that has been specified as an EDEF at link time and placed in the bound unit symbol table. If a symbol name is used, Patch finds the name in the symbol table and uses its address as the base value. The format for the symbol name as a base is +symname, where symname comprises 1 to 12 characters. If a hexadecimal address is used for base, the plus sign is not required.

For bound units created by the MOD 400 Linker the values specified for the /addr fields and IMA references (if any) must include the displacement of the root or overlay. The displacement is equal to the base address of the root or overlay as printed on the link map. The user may add the displacement to each /addr field and IMA, or achieve the same result by specifying the base parameter in the Patch directive. For example, if the first overlay of a bound unit is based at 1000 and a patch to locations 100 to 103 and 200 to 204 is to be made within the overlay, the following two patch directives are equivalent when applied to a LAF bound unit.

```
SP NUMBRA00;/1100/LDR $R1, 1500;STR $R,=$R2

    /1200/ADD $R1, 1600;JMP 1156

SP NUMBRA00;1000;/100;LDR $R1,500;STR $R1,$R2

    /200;ADD $R1,600;JMP 156
```

/addr

Relative location at which the first (or only) subsequent patch value will be applied. Each address must comprise one through six right-justified hexadecimal characters, and must be preceded by the character "/" Subsequent patch values, if any, are applied to succeeding memory locations.

NOTE

Object unit locations can be obtained by examining the listing produced during assembly of the object unit.

instructionn

Value to be inserted at an address, replacing the contents of that location. The value must be specified as:

opcode field$_1$ [,field$_2$] [,field$_3$]

where opcode specifies an Assembly language instruction (except for I/O or floating point instructions); field specifies either a register or a hexadecimal value.

oldval

Specify the current contents of the specified location. If the current contents are not oldval , all patches associated with patchid will not be applied.

NOTE

When using verification patches, specify oldvaln in hexadecimal notation, not as an Assembly language instruction.

VDEF

VDEF

Assign a specified value to an undefined external symbol and change all locations that reference this symbol to the specified value.

FORMAT:

```
VDEF;symname;value [;L]
```

ARGUMENTS:

symname

Name of the external reference that will be assigned a value; can be from 1 to 12 characters in length.

value

Value that is assigned to all references to symname.

[;L]

List all changed references to symname on the device specified as user-out.

Default: No list.

Example:

```
VDEF;VALZZ;50;L
```

Assign the value 50 to the undefined external symbol VALZZ and change all locations that referenced VALZZ to 50.

NOTE

Undefined external references in a bound unit can be defined by a VDEF patch directive only one time. If you make a mistake, you must use HP or DP directives to change each location containing the incorrectly defined value. No listing of the VDEF patch processing is kept; therefore, the L argument should be used.

VDEF is used for changing undefined value definitions. LDEF is used for changing undefined location definitions.

VERIFY/SET PATCH REVISION NUMBER

Allow a revision number to be assigned to a bound unit patch. The revision number may be assigned unconditionally, or on condition that a specified number agrees with the revision number currently in the unit. The patch revision number is stored in the unit as an external value definition with the name ZPTREV.

FORMAT:

VN (str_1, str_2)

ARGUMENTS:

str_1

Character string from one through four hexadecimal digits that is compared with the current patch revision number. If the string does not match the current revision number, no change is made, and Patch terminates.

str_2

Character string from one through four hexadecimal digits to which the patch revision number may be set. If str_1 is omitted or if str_1 matches the current revision number, the patch revision number is set to the value of str_2. If str_1 is omitted and ZPTREV does not exist in the bound or load unit, an external value definition is created with a value of str_2. If str_1 is specified, str_1 and str_2 must be enclosed by parentheses.

NOTE

This directive should not be used when patching an object file.

Appendices

Appendix A
TRAP HANDLING

A trap is a special software or hardware related condition that may occur during execution of a task. Traps include such conditions as a program error, memory defect, arithmetic overflow, or the issuance of an instruction that calls for hardware/software not configured into the system. Table A-1 lists the traps to which the system's hardware/firmware responds.

The design of any application program should provide that when a trap occurs, the hardware/software response will include calling a dedicated software routine (a trap handler) to react to the trap. When trap handlers are provided, the task that caused the trap may now handle the trap in a systematic and orderly way.

TRAP SAVE AREAS

Trap handling routines make use of trap save areas (TSAs). A trap save area is a 104-word data structure that contains the following:

- The contents of several registers. These registers are available for use by the trap handling routine because their contents can be restored upon the routine's completion.
- The instruction associated with the trap.
- The address contained in the program counter when the trap occurred. This is the address to which a return is made when the trap handler routine terminates.
- Two words of additional information related to the trap.
- Trap handler work space.

The number of TSAs built by the system is determined by the value that the user gives the TSA argument of the SYS directive when configuring the system (see the System Building and Administration manual).

Table A-1. Contents of Selected Words of Trap Save Area When Trap Occurs

Trap Number and Condition	Specific Event	Saved Instruction	Saved X-Word[a]	Saved A-Word	Saved Program Counter
0 Cleanup trap	One of the following: • Trap condition for which no other trap is enabled • Abort Task Group command				
1 Monitor call; implicitly handled by the operating system	MCL instruction	0001	80x1	Unspecified	Next location
1 Software trap	PI Command	Unspecified	Unspecified	Unspecified	Unspecified
2 Breakpoint instruction	BRK instruction	0002	90x1	Unspecified	Next location
3 Scientific floating point operation when SIP hardware not in system	Scientific instruction whose address expression generates a reference to a register	Instruction that caused trap	80x1	Unspecified	Unspecified
	Scientific instruction whose address expression generates a reference to memory	First word on instruction that caused trap	00xy	Effective address generated by address expression[b]	Next instruction
4 Unrecognized op code	Instruction not recognized by CPU or SIP	Same as for trap 5	Same as for trap 5	Effective address of trap operand	Next instruction
5 Scientific Branch Instruction when SIP hardware not in system, or any other operation not supported	Undefined instruction whose address expression generates a reference to a register	Instruction that caused trap	80x1	Unspecified	Next instruction
	Undefined instruction whose address expression generates a reference to memory	First word of instruction that caused trap	00xy	Effective address generated by address expression	Next instruction
6 Integer arithmetic overflow (with appropriate overflow trap enable bit of M1 register set to 1)	Overflow of target R-register during execution of instruction whose address expression generates a reference to a register	Instruction that caused trap	80x1	Unspecified	Next instruction
	Overflow of target R-register during execution of instruction whose address expression generates a reference to memory	First word of instruction that caused trap	00xy	Effective address generated by address	Next instruction
7 Scientific divide by zero	A scientific divide (SDV) instruction has a divisor of zero	Unspecified	Unspecified	Pointer to scientific instruction that caused trap	Next CPU instruction
8 Exponential overflow	A scientific operation produces an exponent greater than +63	Unspecified	Unspecified	Pointer to scientific instruction that caused trap	Next CPU instruction

Table A-1 (cont). Contents of Selected Words of Trap Save Area When Trap Occurs

Trap Number and Condition	Specific Event	Saved Instruction	Saved Z-Word [a]	Saved A-Word	Saved Program Counter
13 Unprivileged use of privileged operation	HLT, RTCN, RTCF, WDTN, or WDTF [c] instruction	Instruction that caused trap	8001	Unspecified	Next location
	LEV instruction whose address expression generates reference to a register	Instruction that caused trap	8001	Unspecified	Next instruction
	LEV instruction whose address expression generates a reference to memory	First word of instruction that caused trap	000y	Effective address generated by address expression	Next instruction
	Input/output instruction whose first-word address expression generates a reference to a register	First word of instruction that caused trap	8002	Unspecified	First word of instruction that caused trap, plus 2
	Input/output instruction whose first-word address expression generates a reference to memory	First word of instruction that caused trap	000y	Effective address generated by address expression	First word of instruction that caused trap, plus y
14 Unauthorized reference to protected memory	Instruction that refers to a memory segment with a ring privilege higher than the executing task's	First word of instruction that caused trap	00xy	Effective address generated by address expression	Next instruction
15 Reference to unavailable resource	Instruction whose address expression generates a reference to (1) a memory address higher than the highest memory address available but less than 64K or (2) through indexing, a "wraparound" memory address higher than 64K or less than 0	First word of instruction that caused trap	00xy	Effective address generated by address expression	Next instruction
	Input/output instruction that specifies an improper channel number; address expression generates a reference to a register	First word of instruction that caused trap	008y	Unspecified	First word of instruction that caused trap, plus y
	Input/output instruction that specifies an improper channel number; address expression generates a reference to memory	First word of instruction that caused trap	008y	Effective address generated by address expression	First word of instruction that caused trap, plus y
	WDTN or WDTF instruction and watchdog timer not installed	0006 (WDTN); 0007 (WDTF)	80x1	Unspecified	Next location

Table A-1 (cont). Contents of Selected Words of Trap Save Area When Trap Occurs

Trap Number and Condition	Specific Event	Saved Instruction	Saved Z-Word *	Saved A-Word	Saved Program Counter
16 Program logic error	RTT instruction while TSAP contains a null pointer	Instruction that caused trap	80x1	Unspecified	Next instruction
	Instruction whose address expression illegally generates reference to a register (i.e., this instruction is not permitted to use a register address syllable)	Instruction that caused trap	80x1	Unspecified	Next instruction
17 Bus parity or memory error	Bus parity error or unrecoverable memory data error	Unspecified	Unspecified	Unspecified	Unspecified
19 Scientific underflow	An operation produces an exponent value of less than -64 while the associated enable bit in register M5 is set	Unspecified	Unspecified	Pointer to scientific instruction that caused trap	Next CPU instruction
20 Program error (SIP)	A program error is detected by the SIP	Unspecified	Unspecified	Pointer to scientific instruction that caused trap	Next CPU instruction
21 Scientific significance error	An integer is truncated during floating-point-to-integer conversion while the associated enable bit in register M5 is set	Unspecified	Unspecified	Pointer to scientific instruction that caused trap	Next CPU instruction
22 Scientific precision	The nonzero portion of a fraction is truncated while the associated enable bit of register M5 is set	Unspecified	Unspecified	Pointer to scientific instruction that caused trap	Next CPU instruction
23 Nonexistent resource error	The SIP or Commercial CP attempts a write or read request bus cycle and receives a NAK	Unspecified	Unspecified	Pointer to scientific instruction that caused trap	Next CPU instruction
24 Noncorrectable memory error or Megabus error	A read error occurs which the EDAC cannot correct, or the SIP or CIP detects a parity error	Unspecified	Unspecified	Unspecified	Unspecified
25 Commercial CIP divide by zero	The divisor of a decimal divide instruction (DDV) is equal to zero	Unspecified	Unspecified	Pointer to Commercial CP instruction that caused trap	Next CPU instruction

Table A-1 (cont). Contents of Selected Words of Trap Save Area When Trap Occurs

Trap Number and Condition	Specific Event	Saved Instruction	Saved 2-Word	Saved A-Word	Saved Program Counter
26[d] Commercial CP illegal specification	Any of the following: • Undefined CIP op code detected. • One or both descriptors of an alphanumeric instruction is packed decimal. • Decimal operand has a zero length. • Operand in an Edit, VRF, or SRH instruction has a zero length. • A separate signed decimal operand consists of only a sign (i.e., no digits). • In a Move and Edit instruction, the length of the receiving field was not exhausted, but either there are no micro-ops or the sending field length is exhausted. • A second data descriptor specifies an IMO, except for DCM and ACM instructions. • The first data descriptor in a DSH specifies an IMO. • A third data descriptor specifies an IMO. • In an SRH instruction, the search length list is less than the search argument list, or operand length less than operand element length. • In a VRF instruction, verify list length is less than verify argument length, or operand length is less than operand element length.	Unspecified	Unspecified	Pointer to Commercial CP instruction that caused trap	Next CPU instruction

Table A-1 (cont). Contents of Selected Words of Trap Save Area When Trap Occurs

Trap Number and Condition	Specific Event	Saved Instruction	Saved Z-Word[a]	Saved A-Word	Saved Program Counter
27[d] Commercial CP illegal character	Any of the following: • Illegal decimal digit detected (low-order four bits are not 0 through 9). • Illegal sign digit detected (not a recognized sign value). • Illegal overpunch digit detected.	Unspecified	Unspecified	Pointer to Commercial CP instruction that caused trap	Next CPU instruction
28[d] Commercial CP truncation error	Receiving field of an alphanumeric instruction cannot contain all characters of the result. Whether or not a trap occurs, the receiving field is altered to contain the leftmost part of the result and the CI (TR) is set.	Unspecified	Unspecified	Pointer to CIP instruction that caused the trap	Next CPU instruction
29[d] Commercial CP overflow	Any of the following: • Receiving field of a decimal instruction cannot contain all significant digits of the result. • During a Shift Lift instruction, a nonzero digit is shifted out.	Unspecified	Unspecified	Pointer to CIP instruction that caused the trap	Next CPU instruction
48 Software trap	BREAK command	Unspecified	Unspecified	Unspecified	Unspecified
49 Software trap	UW command	Unspecified	Unspecified	Unspecified	Unspecified
53 Software trap	Resumption of power by automatic power resumption facilty	Unspecified	Unspecified	Unspecified	Next location

[a]The Z-word format is described later in this section.

[b]This is the address of the high-order (leftmost) end of a two-word operand.

[c]If the watchdog timer is not present, this instruction causes a trap to vector 15 regardless of the privilege mode of the central processor.

[d]The Assembly Language Reference manual describes Commercial Central Processor trap handling in detail.

TRAP HANDLING DURING TASK EXECUTION

There are several kinds of traps, as follows:

1. Traps handled by the system exclusively; Monitor Call is currently the only trap of this type.

2. Traps handled first by the system, then possibly by the user. These include Trace/Break if Debug is used, or SIP when the simulator is present.

3. Traps, if enabled, handled by the user program; otherwise, by the system.

4. Software generated traps, described below.

5. Cleanup (trap 0); not really a trap since there is no TSA (trap save area), which is indicated when $B3 is set to null. The condition causing trap 0 is the occurrence of any other trap for which no trap handler has been enabled. $R1 contains the number of the unenabled, causative trap; other registers are unchanged. Return from Trap (RTT) execution is not possible.

In cases 2 and 3 above, which go to the user program, $R3 contains the trap number; $B3 contains a pointer to the TSA.

Software Generated Traps

Software generated traps comprise the following:

- Program Interrupt (trap 1) - Caused by the PI command or signal trap ($SGTRP) macro call.

- Unwind (trap 49) - Caused by the Unwind command.

- Suspend (trap 48) - Caused by the system's break handler, BREAK command, or by the signal trap ($SGTRP) macro call. The system suspends the task when no handler is provided.

- Power resumption notification (trap 53) - Caused by automatic resumption of power after power failure on systems configured with the power resumption facility (described in Appendix E).

To receive the PI, Suspend, or power resumption notification trap, the user program must enable it with the $TRPHD and $ENTRP macro calls.

Program Use of Traps

The average program requires that the trap handler address be set (with the $TRPHD macro call), and that the "cleanup trap" (trap 0) be enabled with the $ENTRP macro call. In more complex situations, requiring more than one cleanup action and, consequently, more than one trap handling routine, the trap handler address can be altered by means of the $TRPHD macro call.

To respond to Program Interrupt (PI), trap 1 must be enabled with the $ENTRP macro call. The trap handler distinguishes between Program Interrupt and cleanup (trap 0), by comparing $B3 with null (see above). In simple programs, for Program Interrupt to resume execution at some other location, the saved P-counter in the trap save area (TSA) must be set, and a Return from Trap (RTT) instruction executed. For more complex programs, the user program should set a flag, then execute a Return from Trap (RTT) instruction. The user program must then examine the flag at appropriate places to avoid interrupts at inappropriate times (e.g., in the middle of a write function).

Alternatively, trap 1 is not enabled; cleanup checks $R1 for X'0301' (the error message signifying that no trap handler exists for a Program Interrupt condition), then branches to the desired location. When cleanup occurs, cleanup (trap 0) is automatically disabled; it may be reenabled when required.

CONTENTS OF TRAP-RELATED MEMORY AREAS

In examining a dump to determine the nature of a trap condition, check particularly the contents of the TSA. The TSA and related memory areas are illustrated in Figure A-1; their contents are described below.

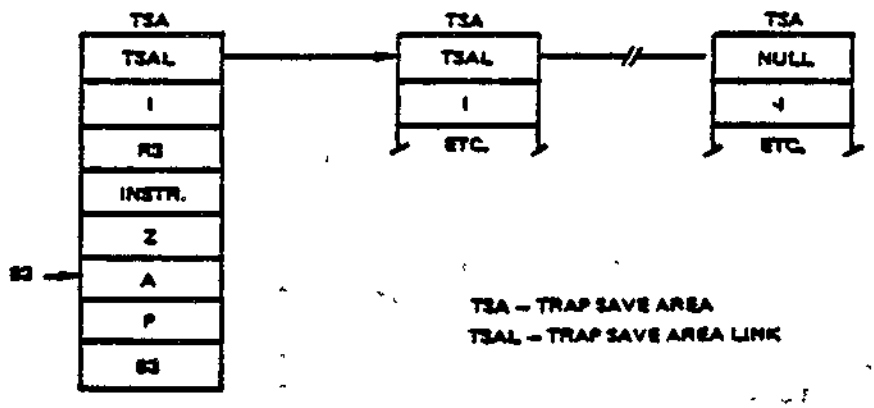


Figure A-1. Trap Handling Mechanism

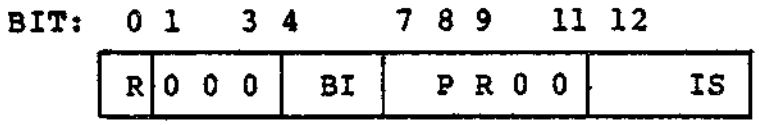
- Trap Save Area Link - When the trap save area is in use, TSAL contains a null pointer (if this is the only or last trap save area connected) or it points to the next trap save area connected. The next TSA connected would be used for handling a trap condition encountered by the trap handling routine (i.e., a nested trap).

- I-Register - The contents of this register are saved by hardware/firmware when a trap occurs. This register is then available for use by the trap handler. The high-order byte contains the quantity (40_{16} - trap number).

- R3 Register - The contents of this register are saved by hardware/firmware when a trap occurs. This register is then available for use by the trap handler.

- Instruction - The hardware/firmware stores the instruction associated with the trap. If a multiword instruction is involved, the first word is saved.

- Z-Word - This word contains miscellaneous information relative to the trap. The format of this word is shown below:



  R - If R=0, the saved contents of the A-word are meaningful relative to this trap condition; if R=1, the saved contents of the A-word are <u>not</u> meaningful.

  BI - 4-bit field that is meaningful only when an indexed bit or byte instruction is associated with the trap. If an indexed <u>bit</u> instruction is involved, BI indicates the four low-order bits of the associated index register; bit 7 of BI stores the least significant bit. If an indexed <u>byte</u> instruction is involved, bit 4 of BI indicates the least significant bit of the associated index register, and bits 5 through 7 are zeros.

  PR - The privilege state of the task that was running when the trap occurred. 00 or 01 = nonprivileged state; 11 or 10 = privileged state. The value is taken from the P-bit of the S-register.

  IS - The length (in words) of the instruction associated with the trap. If a multiword instruction is involved and the trap occurs before the entire instruction has been fetched, IS indicates the number of words that were fetched before the trap.

- A-Word. In many cases, this word contains an address associated with the trap. (This word is not meaningful if bit 0 of the Z-word contains a 1.) The nature of the saved address is governed by the specific trap condition and the specific instruction associated with the trap. Details relative to each trap condition are in Table A-1.

- Program Counter - The contents of the program counter are saved by the hardware/firmware when a trap occurs. This is the address to which a return is made when the trap handler completes. In most cases, the program counter will point to the instruction or location following the instruction associated with the trap. However, when an input/output instruction is involved, the program counter may point to an address within the instruction; in this case, the trap handler must modify this word before issuing a return to "normal" task processing.

- B3 Register - The contents of this register are saved by hardware/firmware when a trap occurs. This register is then available for use by the trap handler; as the trap handler is entered, the B3 register points to the A-word in the trap save area.

SYSTEM SUPPLIED TRAP HANDLERS

The following software components provide trap handling facilities:

- Debug program
- Scientific Simulator
- Defective memory trap handler
- Default trap handler.

Traps handled by these system components can be passed onto user-written trap handlers, as explained later in this section.

Trap Handling by the Debug Program

The Debug program operates as a task within the user group (Multi-User Debugger) or as a task group identified by $D ($D DEBUG). For a detailed description of the Multi-User Debugger and $D DEBUG, see Sections 18 and 17, respectively. In this subsection, both debuggers are referred to collectively as the Debug program.

Once the Debug program is loaded, you may set, clear, or list breakpoints in the task code by use of Debug directives. When the application program is executed, the Debug program is activated by trap number 2, which occurs each time a breakpoint is encountered. The action specified by the Debug directive for that breakpoint will then be executed. For example, designated memory locations can be printed out and execution of the application program continued without operator intervention. Information can be printed on a console or a line printer.

Trap Handling by Scientific Simulator

When a system's configuration does not include a Scientific Instruction Processor (SIP), this hardware component can be simulated by the Scientific Branch Simulator and the Floating Point Simulator, which, together, make up the Scientific Simulator.

FLOATING-POINT SIMULATOR

The Floating-Point Simulator reacts to trap number 3 (scientific operation not in hardware), which occurs whenever the central processor encounters a nonbranch scientific instruction during task processing.

While processing scientific instructions, the simulator provides automatic alignment of the operand's hexadecimal mantissas. It achieves maximum available precision by requiring that mantissas have no leading zeros (i.e., all mantissas must be normalized).

Note the following programming consideration for the simulator:

- During its processing, the simulator may encounter an error condition related to a scientific instruction; the following can then occur:
  - The simulator consults trap vector 5 if it encounters a nonscientific instruction or other unrecognized instruction.
  - The simulator consults trap vector 7 if an SDV (Scientific Divide) instruction has a divisor of 0. The instruction will not be executed.
  - The simulator consults trap vector 8 if execution of a scientific instruction produces exponential overflow. The instruction will have been executed.
- To use a software routine to react to any of these trap conditions, the user must provide a user-written trap handler. The simulator will be invoked to handle traps caused by execution of scientific instructions only if the trap numbers have been enabled for the task executing those instructions.
- No "overflow trap enable" bit of the M1 register should be set to 1 as the simulator begins operation.

SCIENTIFIC BRANCH SIMULATOR

The Scientific Branch Simulator reacts to trap 5. It provides FORTRAN and Assembly language programs with the means to simulate the use of the scientific branch instructions.

Note the following programming considerations relative to the simulator:

- The choice of the single-precision version (SSIP), or the double-precision version (DSIP) of the simulator is indicated in an argument of the system building SYS directive.

For SSIP only:

- The simulator uses registers R4, R5, and R7 as scientific accumulator (S1) for comparisons; it uses R1, R2, and R3 as work registers.

- The simulator uses the G, L, and U bits of the I register to determine if the branch condition is true or false. When a normal return is made to the user program, the branch will be executed if the branch condition is true; otherwise, the next sequential instruction following the one that was trapped will be executed.

For both SSIP and DSIP:

- All other operation codes not handled by the Floating-Point Simulator or the Scientific Branch Simulator are passed to the next trap handler in trap 5.

Defective Memory Trap Handler

The defective memory trap handler performs the following:

- Identifies to the user the physical and virtual address of defective memory.

- Informs the user whether or not the system remains operable after the detection of defective memory.

- Ensures that the area of defective memory will not be reallocated after its detection.

The user loads the defective memory trap handler at the time of system configuration by entering the LDBU directive and specifying the simple pathname to the bound unit ZXDEFM (see the System Building and Administration manual).

The defective memory trap handler responds to detection of defective memory by the following components:

- Central Processor Unit
- Scientific Simulator
- Input/Output controller.

If defective memory is detected by any of these three components, and the system is able to continue, the following message is sent to the operator's console, specifying the physical and virtual address of the defective memory:

PROBABLE MEMORY FAILURE, PHYSICAL ADDR= ,VIRTUAL ADDR=

If the defective memory is CPU-detected (trap 17) and no user-written trap was enabled for trap 17, an X'0311' error message is also issued and the trapped program terminates.

If a user-written trap handler is enabled for trap 17, the defective memory trap handler ensures that the 32-word area containing the defective memory will not be reallocated to another task, and control is passed to the user-written trap handler, which normally returns task resources and terminates the task request.

If the defective memory is detected by the Scientific Simulator (trap 24), and, if no user-written trap handler is enabled for trap 24, the X'0318' error message is issued (see the System Messages manual) and the trapped program terminates.

A defective memory trap resulting from a file system I/O order produces the probable memory failure message followed by an X'0107' error message (see the System Messages manual).

If defective memory is detected, and the system is unable to continue, register contents are as follows:

$R1 - X'DEFA' (defective memory address)
$B1 - physical address of defective memory
$B2 - virtual address of defective memory

Knowledge of the address of defective memory permits the user to map the defect onto a specific memory board, which can then be replaced.

Whenever memory is found defective, it is returned to the memory manager and marked as unavailable for reallocation. Before memory can be returned to the memory manager, it must be relinquished by all of its users. For that reason, if memory found defective is within a shared area, such as a sharable bound unit or group control block, each task sharing that memory is liable to be trapped and terminated.

When defective memory is marked unavailable for reallocation, at least 32 words are so marked. Trap 17 and 24 identify the exact location of memory detected as defective. I/O controller detection is less precise since it knows that only some location within the buffer is defective. In this case the memory manager makes unavailable all pages containing any part of the suspect buffer. The address cited in the probable memory failure error message is the beginning of the suspect buffer.

System Default Trap Handling

When a trap condition occurs in task code that has not enabled this particular trap or trap 0, an error message is written to the error-out file; the delete bit in the task control block is reset, the task is terminated, but the task's resources (memory and peripherals) are not released. Thus, a memory dump can be taken so that the error condition can be examined.

USER-WRITTEN TRAP HANDLERS

User-written trap handlers are either task-specific or system-wide. Both types are described below.

Task-Specific Trap Handlers

This type of trap handler is included in a task's bound unit; it resides in a task group's memory pool. A task-specific trap handler receives a trap only if the task, in whose bound unit the handler is included, has done the following:

- Specified the trap number, by means of the Enable User Trap ($ENTRP) macro call.
- Connected the trap handler to the trap's vector by means of the Trap Handler Connect ($TRPHD) macro call.

The task-specific handler receives the TSA contents exactly as if it was directly connected to the trap vector; but, in fact, the monitor has intercepted the trap and simulated the TSA in user-accessible memory.

System-Wide Trap Handlers

A system-wide trap handler is loaded into system memory at the time of configuration. It is directly attached to a specific trap vector by user code. When any executing task in the system signals that trap, the trap handler directly responds, bypassing the Monitor (which, for a task-specific trap handler, would intercept and analyze the trap). Thus, system overhead is reduced; however, the same trap handling routine services all tasks that incur a given trap condition.

PASSING TRAPS

It is assumed that all vendor-supplied and possibly some user-written trap handlers attached to the vector may encounter situations which should be passed to the system default trap handler. Also, several handlers can process the same trap. To pass a trap from one handler attached to a trap vector to the next handler:

1. Load the trap handler by means of an LDBU directive, thus placing the handler in system memory. The system, at the time of configuration, implicitly loads the Scientific Simulator's trap handler into system memory if the SSIP or DSIP argument was specified in a SYS directive.

2. Write the handler to include initialization subroutine table (IST) code that will execute when the LDBU load operation occurs and save the current address contents of the trap vector(s) to be simulated, inserting its own pointer(s) instead.

3. Code the user-written simulator to save the contents of all registers upon entry so that if the trap should be passed to the next trap handler, this handler can:

   a. Restore all saved registers.

   b. Execute a jump-indirect through the location containing the pointer of the next handler saved in step 2 above. The J-bit in the M1 register must be off when the jump-indirect is executed.

The rule is that each trap handler must get exactly the same information in registers and TSA that it would have received if it was the first trap handler accessed.

Programming Considerations for User-Written Trap Handlers

- A trap handler operates at the same priority level and in the same privilege ring as the task whose execution caused the trap.

- When a trap occurs, the hardware/firmware saves the task related contents of the I-register, the R3 register, and the B3 register in the trap save area. The trap handler is free to use these registers.

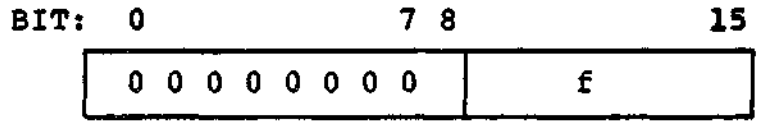
- See Table A-1 for a description of the contents of selected words in the trap save area when various traps occur.

- Upon entry to the user trap handler, the J-bit in the M1 register is arbitrarily turned off. Other bits in the M1 register remain as they were when the trap occurred. Register B3 contains a pointer to the A-word in the TSA. Register R3 contains the vector number of the trap.

- Traps that occur within the user trap handler abort the task if they are the same type as the trap currently being processed. This abort action prevents all TSAs from being tied up by recursive traps, and prevents traps within the MCL interface from going to the user trap handler.

- Every trap handler should be reentrant; i.e., it should not use an internal work area to store interim information, since this information could be lost if an interrupt occurs and, later, the same trap handler is called upon to execute at a different priority level.

- If you choose to define instructions of your own and have them interpreted by a trap handler connected to trap vector 5, you should limit the instructions to the user-reserved subset of the generic instructions. The following diagram illustrates the memory format of generic instructions.



f - Function; the user-reserved range of values for f is $128 \leq f < 256$ (decimal).

- When a trap handler has finished its work, it must issue an RTT (Return From Trap) instruction. The M1 register is not restored. This instruction uses the current trap save area to restore the task-related contents of the I-register, the R3 register, the program counter, and the B3 register. Consequently, when the RTT instruction is executed, these elements of the trap save area should be "correct" (i.e., as saved when the trap occurred).

  Note that in some cases, particularly when a trap condition is related to an input/output instruction, the saved value of the program counter (in the trap save area) will point to a memory location <u>within</u> the instruction itself. This is <u>not</u> a legitimate point of return to "normal" task processing. In this case, the trap handler must modify the saved value of the program counter before issuing an RTT instruction.

  After the trap save area has been used to restore the registers indicated above, it is returned to the pool of available trap save areas pointed to by a memory location 0010.

- When a trap occurs, the contents of registers M1 through M7 are not saved in the TSA. Particular attention is drawn to the R1 through R7 overflow trap enable bits and the J-bit of register M1, which can be set by a privileged user. If the trap handler does not temporarily clear these bits during its execution, another user trap handler could be invoked erroneously on data register overflow or branches. Such bits must be restored upon exit from the handler.

Appendix B
PROGRAMMING CONVENTIONS

The following programming conventions are provided for designing application programs to interface smoothly with system software.

MODULE AND FILE NAME CONVENTIONS

Program names and load module names that begin with Z are reserved for Honeywell use and should not be used for an application program. System module names are six characters in length; the second character defines the system component. Table B-1 lists the first two characters of each system module name and the system component that it relates to.

The names of files that are processed by program development software (compiler, assembler, and so on), are given a suffix by the particular component doing the processing. Table B-2 lists these suffixes.

Table B-1. System Module Name-Prefixes

Name Prefix	System Component
ZA	Assembler
ZC	COBOL Compiler
ZE	Editor
ZF	FORTRAN Compiler
ZG	Configuration Load Manager
ZH	Trap Handler
ZI	Input/Output Drivers
ZL	Linker
ZM	Memory Management
ZO	Loader
ZP	Macro-Assembly Program
ZQ	Communications
ZR	RPG Compiler
ZS	Sort/Merge
ZT	TCLF Compiler and Processor
ZU	Utility Routines and Conversion Aids
ZX	Executive
ZY	File, Data and Storage Management
ZZ	Program units internal to File, Data and Storage Management
Z1	Advanced FORTRAN Compiler

Table B-2. System Program File Name Suffixes

Suffix	File Type
.A	Assembly language source unit
.AO	Default user-out if user-in is disk
.B	BASIC source program unit
.C	COBOL language source unit
.DB	Multi-user Debugger work file
.EC	Execution command (EC)
.F	FORTRAN language source unit
.L	List unit
.M	Link maps
.O	Object unit
.P	Macro-Assembly Program source program unit
.PS	PASCAL source unit
.Q	RPG Compiler generated linker directive file
.QK	Multi-user Debugger quick file
.R	RPG language source unit
.T	TCLF source program unit
.U	Auto report source unit

CALLING SEQUENCE FOR EXTERNAL PROCEDURES

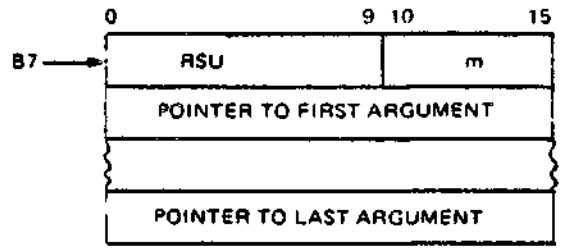
External procedures are those that are assembled or compiled separately from the calling procedure. These procedures may be either functions, that is, procedures returning a single value to the caller, or subroutines, namely, procedures that alter data contained in an area common to both the procedure and its caller. For example, the FORTRAN mathematical routines (sine, cosine, etc.) are external procedures. When it is necessary to write an Assembly language external procedure, use the calling sequence described below for compatibility with code generated by the language processors.

The external procedure calling sequence generated by the CALL statement in Assembly language, COBOL, BASIC, FORTRAN and RPG is of the form:

```
LAB $B7, list
LNJ, $B5.<entry
```

list - Label assigned to the argument list
entry - External label of subroutine's entry point

The external procedure should assume that register B5 contains the address of the caller's return point and register B7 points to an argument list having the format shown in Figure B-1.



RSU: Reserved for system use (must not be modificed by called procedure)
m: Length of argument list given by $SAF*n+1 where n is the number of arguments

Figure B-1. Argument List

REGISTER CONVENTIONS

The system services use the following registers without preserving their contents: R1, R2, R6, R7, B2, and B4. If the information in these registers is of value to the application program, it should save the register contents before making a system control service request. Unless otherwise specified, the following registers will not be altered by the system services: S, I, R3, R4, R5, B1, B3, B5, B6, B7, T, RDBR, CI, SI, S1, S2, S3, and the M registers.

Appendix C
ASSEMBLING, LINKING, AND EXECUTING A PROGRAM

This appendix describes procedures assembling, linking, and executing an Assembly language source program.

INTRODUCTION

Assembly language programs are assembled by means of the Macro-Assembly program (MAP), which processes macro calls and assembles the source unit in one pass.

Input to MAP consists of a source program written in Assembly language and optional control information. Output from MAP is:

- An Assembly language object (.O) program
- An Assembly language listing and diagnostic.

Input to the Linker consists of the relocatable object program. Output from the linker is:

- An executable module
- A link map.

Figure C-1 illustrates the operation of assembling (by means of MAP) and linking, which produces an excutable bound unit.

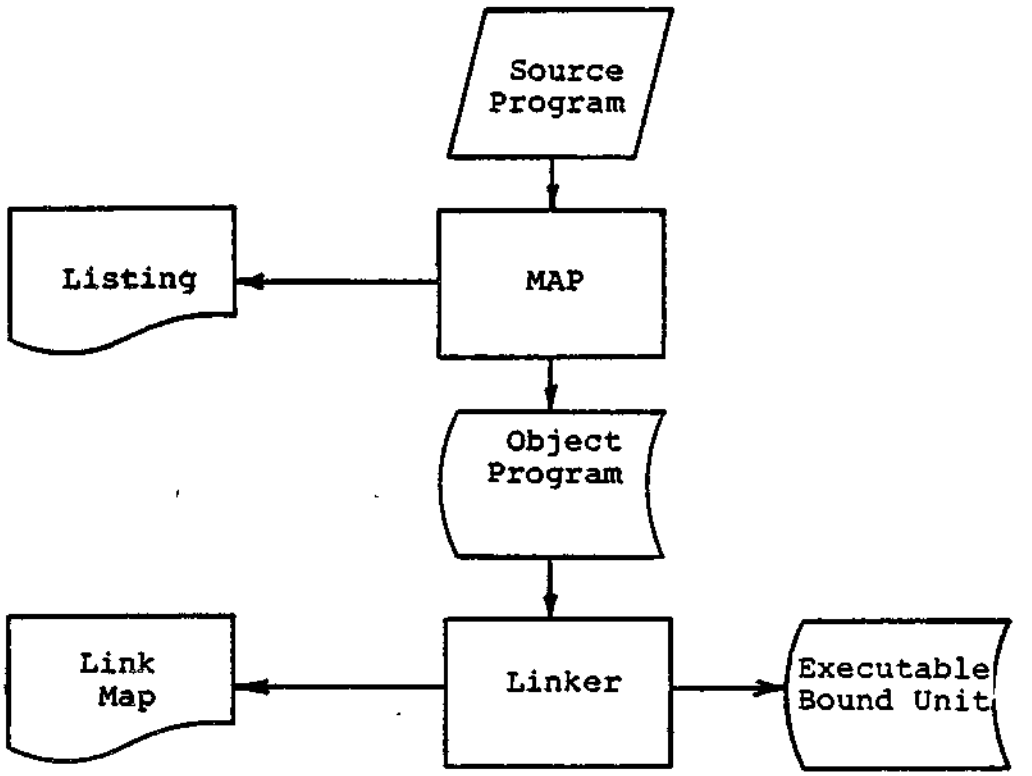


Figure C-1. Assembling and Linking a Program.

INVOKING MAP

To assemble an Assembly language source program by means of the MAP facility, enter the following command:

```
MAP  path  [ctl_arg]
```

where:

path — The pathname of the input source file. MAP appends a .A to the supplied pathame; if this file is not found, MAP appends a .P to the supplied pathname to locate the source input file. Do not append any suffix to the pathname when supplying it in the command line.

ctl_arg — None or any number of control arguments, which are fully described in the Commands manual.

Example:

The source file to be assembled is ADD.A, shown in Figure C-2. This file is in the current working directory. MAP is invoked by the following command line:

```
MAP  ADD
```

The terminal dialog is:

```
MAP ADD                                  Invoke MAP

  MAP-1.1 -07/12/0716                    MAP responds with
                                         version number, date
                                         and time
0000 ERROR COUNT MACRO  :SAMPLE          Macro calls processed
                                         without errors
0000 ERROR COUNT ASSEMBLER  :SAMPLE      Program assembled
                                         without  errors
0000 WARNINGS:  SAMPLE                   No warnings
```

In this dialog, MAP refers to the program by its title, SAMPLE. The title of an Assembly language program is declared in the program's first line, as seen in Figure C-3. Although in this case the program's title (SAMPLE) and simple name (ADD) are different, they can be the same.

MAP produces an object unit (.O suffix) and a listing file (.L suffix). The listing is shown in Figure C-3.

```
          TITLE     SAMPLE,'9-16',ADDING 5 NUMBERS
          LIBM      EXEC.LIB
*
*ADDS NUMBERS FROM LIST AND SUMS THEM INTO $R5
*
*DEFINITIONS
*
*
TABLE     DC        Z'30313233343536373839414243444546'     ASCII TABLE
LIST      DC        1,10,50,100,0                  DATA LIST
ANSWER    DC        A '       STARTING ADDRESS OF ANSWER
ANSNO     DC        0,0       CONVERTED ANSWER
*
*INTERNAL SUBROUTINE
*
ASCII     CL        =$R3      CLEAR INDEX FOR ANSNO
          LDV       $R2,4     LOAD COUNTER
AGAIN     CL        =$R4      CLEAR $R4 TO 0
          DEC       =$R2      DECREMENT COUNTER
          BCF       <DONE     CHECK FOR END OF LOOP
          DOL       $R5,4     MOVE FIRST 4 BITS INTO $R4
          LLH       $R1,=$R4  MOVE FIRST FOUR BITS INTO INDEX REGISTER FOR TABLE
          LLH       $R6,<TABLE.$R1       MOVE ASCII CONVERSION INTO $R6
          STH       $R6,<ANSNO.$R3       STORE ASCII CONVERSION IN ANSNO
          INC       =$R3      INCREMENT ASNO INDEX
          JMP       <AGAIN    GO TO START OF LOOP
DONE      JMP       $B5       RETURN TO MAIN PROGRAM
*
*INITIALIZATION
*
SETUP     CL        =$R1
          CL        =$R2
          CL        =$R3
          CL        =$R4
          CL        =$R5
          CL        =$R6
          CL        =$R7
          LDV       $R1,5     VALUE FOR COUNTER
*
*BEGIN ADDING
*
START     DLC       =$R1
          BCF       <CONVRT   I(C) IS SET TO 0 WHEN $R1 = -1
          LDR       $R3,<LIST.$R2        GET A NUMBER FROM LIST, USE $R2 AS INDEX
          INC       =$R2      TO MOVE THROUGH LIST
          ADD       $R5,=$R3  ADD VALUES IN TWO REGISTERS
          JMP       <START    LOOP UNTIL BCF BRANCHES
CONVRT    LNJ       $B5,<ASCII
FINIS     $USOUT    'ANSWER,=6           MACRO TO DISPLAY ANSWER, SLEW BYTE
          $TRMRQ    =0        MACRO TO TERMINATE TASK REQUEST
          END       SAMPLE,SETUP         ENTRY POINT IS SETUP
```

Figure C-2. Source Unit ADD.A

```
SAMPLE 9-16      ,ADDING 5 NUMBERS   -LAP  1982/08/06 1424:08.1 MAP-1.1 -07/12/0716  GC0S6 M0D400-L3.0-06/10/1621   PAGE 0001
    000001                                    TITLE     SAMPLE,'9-16',ADDING 5 NUMBERS
    000002                                    *
    000003                                    *ADDS NUMBERS FROM LIST AND SUMS THEM INTO $R5
    000004                                    *
    000005                                    *DEFINITIONS
    000006                                    *
    000007                                    *
    000008  0000  3031                        TABLE     DC        Z'30313233343536373839414243444546'     ANSI TABLE
            0001  3233
            0002  3435
            0003  3637
            0004  3839
            0005  4142
            0006  4344
            0007  4546
    000009  0008  0001                        LIST      DC        1,10,50,100,0             DATA LIST
            0009  000A
            000A  0032
            000B  0064
            000C  0000
    000010  000D  4120                        ANSWER    DC        'A '      STARTING ADDRESS OF ANSWER
    000011  000E  0000                        ANSNO     DC        0,0       CONVERTED ANSWER
            000F  0000
    000012                                    *
    000013                                    *INTERNAL SUBROUTINE
    000014                                    *
    000015  0010  8753                        ASCII     CL        =$R3      CLEAR INDEX FOR ANSNO
    000016  0011  2C04                                  LDV       $R2,4     LOAD COUNTER
    000017  0012  8754                        AGAIN     CL        =$R4      CLEAR $R4 TO 0
    000018  0013  8802                                  DEC       -$R2      DECREMENT COUNTER
    000019  0014  0680 0000 0023                        BCF       (DONE     CHECK FOR END OF LOOP
    000020  0017  5084                                  DBL       $R5,4     MOVE FIRST 4 BITS INTO $R4
    000021  0018  9204                                  LLN       $R1,=$R4  MOVE FIRST FOUR BITS INTO INDEX REGISTER FOR TABLE
    000022  0019  E290 0000 0000                        LLN       $R6,(TABLE,$R1      MOVE ASCII CONVERSION INTO $R6
    000023  001C  E780 0000 000E                        STN       $R6,(ANSNO,$R3      STORE ASCII CONVERSION IN ANSNO
    000024  001F  8A03                                  INC       =$R3      INCREMENT ASNO INDEX
    000025  0020  0300 0000 0012                        JMP       (AGAIN    GO TO START OF LOOP
    000026  0023  8305                        DONE      JMP       $85       RETURN TO MAIN PROGRAM
    000027                                    *
    000028                                    *INITIALIZATION
    000029                                    *
    000030  0024  8751                        SETUP     CL        =$R1
    000031  0025  8752                                  CL        =$R2
    000032  0026  8753                                  CL        =$R3
    000033  0027  8754                                  CL        =$R4
    000034  0028  8755                                  CL        =$R5
    000035  0029  8756                                  CL        =$R6
    000036  002A  8757                                  CL        =$R7
    000037  002B  1C05                                  LDV       $R1,5     VALUE FOR COUNTER
    000038                                    *
    000039                                    *BEGIN ADDING
    000040                                    *
    000041  002C  8801                        START     DEC       =$R1
    000042  002D  0680 0000 0038                        BCF       (CONVRT   I(C) IS SET TO 0 WHEN $R1 = -1


SAMPLE 9-16      ,ADDING 5 NUMBERS   -LAP  1982/08/06 1424:08.1 MAP-1.1 -07/12/0716  GC0S6 M0D400-L3.0-06/10/1621   PAGE 0002
    000043  0030  8820 0000 0008                        LDR       $R3,(LIST,$R2       GET A NUMBER FROM LIST, USE $R2 AS INDEX
    000044  0033  8AD2                                  INC       =$R2      TO MOVE THROUGH LIST
    000045  0034  DA53                                  ADD       $R5,=$R3  ADD VALUES IN TWO REGISTERS
    000046  0035  8380 0000 002C                        JMP       (START    LOOP UNTIL BCF BRANCHES
    000047  0038  0380 0000 0010             CONVRT,    LNJ       $85,(ASCII
    000048                 0038               FINIS     EQU       $
    000049  003B  CHC0 FFD1                              LAR       $84,ANSWER
    000050  003D  E070 0006                              LDR       $R6,=6              SET $R6 TO BUFFER SIZE
    000051  003F  7C00                                   LDV       $R7,0               SET $R7 TO INDICATE LEFTMOST BYTE
    000052  0040  0001                                   HCL
    000053  0041  0801                                   DC        X'0801'             WRITE TO USER OUT FILE
    000054  0042  AB70 0000                              LDR       $R2,=0              SET $R2 TO COMPLETION STATUS
    000055  0044  0001                                   HCL
    000056  0045  0103                                   DC        X'0103'             TERMINATE WITHOUT MODIFYING START ADDRESS
    000057  0046         0024                           END       SAMPLE,SETUP        ENTRY POINT IS SETUP
0000 ERROR COUNT MACRO  =SAMPLE
0000 ERROR COUNT ASSEMBLER  =SAMPLE
0000 WARNINGS   =SAMPLE
00300 WORD SYMBOL TABLE
EOF
```

Figure C-3. MAP Listing of ADD.L

INVOKING THE LINKER

Once the source program is assembled, it can be linked. To invoke the Linker, enter the following command:

```
LINKER progname  [ctl_arg]
```

where:

progname — The bound unit pathname (simple, relative, or absolute) of the bound unit to be created (usually the program name; may be up to 62 characters in length).

ctl_arg — Linker control arguments include -PT, which requests the Linker to issue a prompt (L?) for input. For information on other Linker control arguments, see Section 16.

For example, to invoke the Linker for ADD (assembled above), enter:

```
LINKER ADD -PT
```

The following Linker dialogue results:

```
LINKER ADD -PT              Invoke the Linker.
LINKER -0300-06/18/0912     Linker responds with version
                            number, date, and time.
L?                          Linker prompts for input.
LINK ADD                    Link the object unit.
L?
QUIT                        The last linker directive entered.
ROOT ADD                    Add is linked.
LINK DONE                   The Linker is finished.
RDY:                        Control returns to command level.
```

The Linker produces executable code.

EXECUTING AN ASSEMBLY LANGUAGE PROGRAM

After you have prepared, assembled, and linked your program, simply type in the program name to run it. For our sample program, ADD, successful execution is shown below:

```
ADD         Invoke the executable module.
00A1        Sum is given in hexadecimal.
RDY:        Control returns to command level.
```

Appendix D
DATA STRUCTURE FORMATS

This appendix describes the following data structures:

- Clock request block (CRB)
- File information block (FIB)
- Input/output request block (IORB)
- Task request block (TRB)
- Parameter block
- Wait list
- Semaphore request block (SRB)
- Message group request blocks (MGCRB, MGIRB, MGRRB).

Any of the structures can be hand coded or generated by macro calls. All structures but the parameter block and wait list can be defined by macro call templates.

The first four items of the request blocks have an identical format (but slightly different contents, depending on the block type) as shown in Figure D-1. Later diagrams show the format of each block type; tables show the contents of the block entries.

The offset symbol $AF signifies that number of words required to specify a memory address. In this system, $AF is equivalent to two words.

The first field (-$AF or -1) of a request block need be present only when the request block pointer/semaphore name is needed.

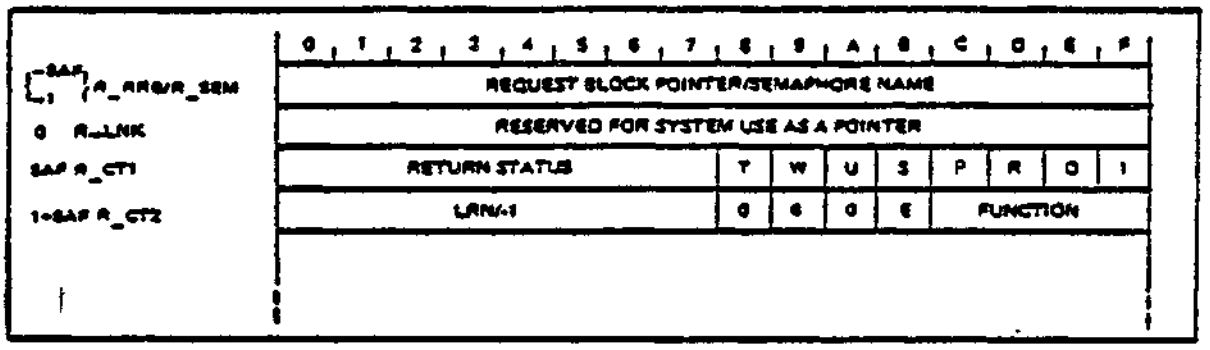


Figure D-1. First Four Items of Request Blocks

CLOCK REQUEST BLOCK FORMAT

Figure D-2 shows the format of the clock request block; Table D-1 shows its contents.

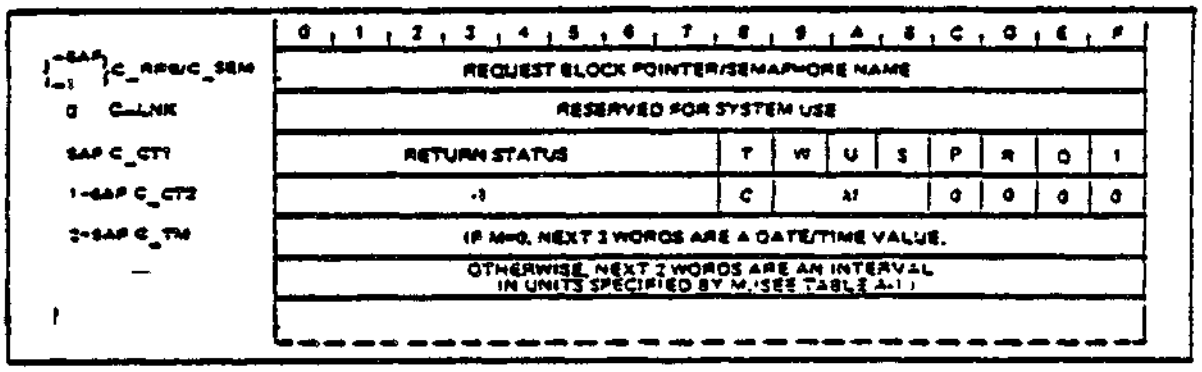


Figure D-2. Format of Clock Request Block

Table D-1. Contents of Clock Request Block

Word	Label	Bit(s)	Contents
-$AF -1	C_RRB/ C_SEM	0-31 0-15	Depending on the condition of the S- or R-bits of C_CTl, this field contains a 2-word task request block pointer (R-bit on), or a 1-word semaphore name (S-bit on).
0	C_LNK	0-15	Reserved for system use.
$AF	C_CTl	0-7	Return status.
		8(T)	This bit is set on while the request using this block is executing; it is reset when the request terminates. The system controls this bit; user should not change it.
		9(W)	Wait bit. Set if the requesting task is not to be suspended pending the completion of the request that uses this block.
		A(U)	User bit. User may or may not use this bit; the system does not change it. In a user-built CRB, must be 0 initially.
		B(S)	Release semaphore indicator. 0 = No release; 1 = Release, on completion of this request, semaphore item named in C_SEM.
		C(P)	Must be set by user if CRB is to be referenced by a Wait Any ($WAITA) macro call. If set, CRB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D(R)	Return clock RB indicator. 0 = No dispatch; 1 = Dispatch task request block named in C_RRB after completion of this request.
		E(D)	Delete clock RB indicator, used usually with the B(S) and D(R) bits. 0 = No delete; 1 = Delete and, when task terminates, return memory to pool where CRB is first entry of its memory block.

Table D-1 (cont). Contents of Clock Request Block

Word	Label	Bit(s)	Contents
$AF (cont)	C_CT1	F	Implicit task start address. Must always be 1 for CRB.
1+$AF	C_CT2	0-7	Value is -1.
		8(C)	When set, indicates this block is associated with a cyclic clock function.
		9-B(M)	When set, last two words contain an interval in units specified by M. Each interval value is as follows: 001 - in milliseconds; 010 - in tenths of a second; 011 - in seconds; 100 - in minutes; 101 - in units of clock resolution.
			When reset (off), the last three words contain a date/time interval.
2+$AF	C_TM		Contents depend on M bit of C_CT2.

FILE INFORMATION BLOCK (FIB) FORMAT AND CONTENTS

Tables D-2 and D-3 show the format, and Tables D-4 and D-5 show the contents, of the file information block (FIB) for data management (record level) access, and for storage management (block level) access, respectively.

Table D-2. Format of FIB for Data Management

Word	Label(s)	0 1 2 3 4 5 6 7 8 9 A B C D E F
0	F_LFN	Logical file number (LFN)
1	F_PROV	Program view
2	F_URP	User record area pointer
3		
4	F_IRL	Input record length
5	F_ORL	Output record length

Table D-2 (cont). Format of FIB for Data Management

Word	Label(s)	0 1 2 3 4 5 6 7 8 9 A B C D E F
6	F_IRS/F_ORS	Input record status | Output record status
7	F_IRT	Input record type
8	F_ORT	Output record type
9 10	F_IKP	Input key pointer
11	F_IKF/F_IKL	Input key format | Input key length
12 13	F_ORA	Output record address
14 15	F_RFU2	Reserved

Table D-3. Format of FIB for Storage Management

Word	Label(s)	0 1 2 3 4 5 6 7 8 9 A B C D E F
0	F_LFN	Logical file number (LFN)
1	F_PROV	Program view
2 3	F_UBP	User buffer pointer
4	F_BFSZ	Buffer size
5	F_BKSZ	Block size
6 7	F_BKN1 F_BKN2	Block number

Table D-3 (cont). Format of FIB for Storage Management

Word	Label(s)	0 1 2 3 4 5 6 7 8 9 A B C D E F
8	F_RFU3	Reserved
9		
10		
11		
12		
13		
14		
15		

Table D-4. Contents of FIB for Data Management

Word	Label	Bit(s)	Contents
0	F_LFN	0-15	Logical file number (LFN)
1	F_PROV	0	Access level. Set off for data management.
		1-4	Process rules. Bit 1 for $RDREC, bit 2 for $WRREC, bit 3 for $RWREC, bit 4 for $DLREC.
		5-9	Key type. Bit 5 for primary keys, bit 8 for relative keys, bit 9 for simple keys (bits 6 and 7 must be 00).
		10	Record class. Set on for fixed-length records only; off for fixed- and variable-length records.
		11	Record visibility. Set on if deleted records are to be visible; off if invisible.
		12	Key storage alignment. Set on if storage area begins at odd-byte boundary; off if even-byte boundary.

Table D-4 (cont). Contents of FIB for Data Management

Word	Label	Bit(s)	Contents
1 (cont)	F_PROV (cont)	13	Record storage area. Set on if record storage area begins on odd-byte boundary; off if even-byte boundary.
		14	Transcription mode. Set on if data transferred in binary transcription mode; off if ASCII mode.
		15	Must be 0.
2,3	F_URP	0-31	Start address of user record area.
4	F_IRL	0-15	Input record length (in bytes).
5	F_ORL	0-15	Output record length (in bytes).
6	F_IRS	0-3	0000 - Unknown terminal control information; 0001 - Records contain no terminal control information; 0010 - Records contain standard GCOS 6 printer control characters.
		4-7	Must be zero.
	F_ORS	8	Read operations. Set on if the key of the record just read duplicates the key of the record previously read. Write/rewrite operations. Set on if the key of the record just written is a duplicate.
		9	Read operations. Set on if the key of the record just read duplicates a record that is yet to be read.
		10-15	Must be zero.
7	F_IRT	0-15	Must be set to X'FFFF' (all bits set on).
8	F_ORT	0-15	Must be set to X'0000' (all bits set off).
9,10	F_IKP	0-31	Start address of user key area.
11	F_IKF	0-7	Input key format. 0 for none specified; 1 for primary key; 2 for simple key.

Table D-4 (cont). Contents of FIB for Data Management

Word	Label	Bit(s)	Contents
	F_IKL	8-15	Input key length (in bytes).
12,13	F_ORA	0-31	Output record address.
14,15	F_RFU2	0-31	Reserved for later use; must be X'00000000'.

Table D-5. Contents of FIB for Storage Management

Word	Label	Bit(s)	Contents
0	F_LFN	0-15	Logical file number (LFN).
1	F_PROV	0	Access level. Set on for storage management.
		1-2	Process rules. Bit 1 for $RDBLK; bit 2 for $WRBLK.
		4-12	Must be X'00000000'.
		13	Buffer alignment. Set on when buffer begins on odd-byte boundary; off when even-byte boundary.
		14	Transcription mode. Set on when data transferred in binary transcription mode; off when transfer is in ASCII mode.
		15	Synchronous/asynchronous indicator. Set on when $RDBLK and $WRBLK calls executed asynchronously; off when synchronously.
2,3	F_UBP	0-31	Start address of user buffer area.
4	F_BFSZ	0-15	Buffer transfer size (in bytes).
5	F_BKSZ	0-15	Block size (in bytes).
6,7	F_BKNO	0-31	Block number.
8-15	F_RFU3	All	Reserved for later use; must be all zeros.

INPUT/OUTPUT REQUEST BLOCK (IORB) FORMAT

Figure D-3 shows the format of a nonextended input/output request block (IORB) (see Section 4 for a description of of IORB extensions). Table D-6 defines the specific fields for a non-extended IORB. Table D-7 summarizes the IORB fields for operator interface functions.

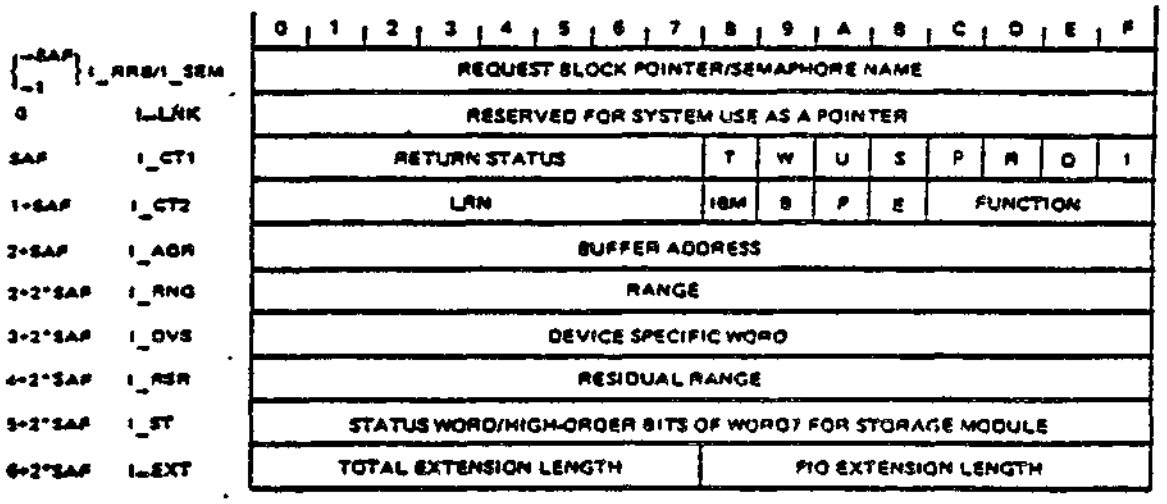


Figure D-3. Format of I/O Request Block

Table D-6. Contents of I/O Request Block

Word	Label	Bit(s)	Contents
-$AF -1	I_RRB/ I_SEM	0-31 0-15	Depending on the S- or R-bits of I_CT1, this field contains a 2-word task request block pointer (R-bit on), or a 1-word semaphore name (S-bit on). Set by user; used by system at termination of request.
0		0-31	Reserved for system use. 2-word pointer to indirect request block.
$AF	I_CT1	0-7 8(T)	Return status This bit is set (on) while the request using this block is executing; it is reset when the request terminates. The system controls this bit; user should not change it.

Table D-6 (cont). Contents of I/O Request Block

Word	Label	Bit(s)	Contents
$AF (cont)	I_CT1 (cont)	9(W)	Wait bit. Set by user if the requesting task is not to be suspended pending completion of the request that uses this IORB.
		A(U)	User bit. User may or may not use this bit; the system does not change it. 0 = No release; 1 = Release, on completion, semaphore item named in I_SEM.
		B(S)	Release semaphore indicator.
		C(P)	Must be set by user if IORB is to be referenced by a Wait Any ($WAITA) macro call. If set, IORB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D(R)	Return IORB indicator. 0 = No dispatch; 1 = Dispatch task request block named in I_RRB after completion of this request. If 1, system executes $RQTSK, using I_RRB, when the task terminates.
		E(D)	Delete IORB indicator. Used usually with the B(S) and D(R) bits. 0 = No delete; 1 = Delete and when task terminates, return memory to pool where IORB is first entry of its memory block.
		F	Implicit task start address. Must always be 1 for IORB.
1+$AF	I_CT2	0-7	Logical resource number (LRN). Identifies device to be used.
		8(IBM)	IBM-type request. Changes interpretation of I_DVS to task word, and of I_RSR and I_ST to configuration words A and B, respectively.

Table D-6 (cont). Contents of I/O Request Block

Word	Label	Bit(s)	Contents
1+$AF (cont)	I_CT2 (cont)	9(B)	Byte index. 0 = buffer begins in leftmost byte of word; 1 = buffer begins in rightmost byte.
		A(P)	Private space; reserved for system use.
		B(E)	Extended IORB indicator. 0 = Standard (nonextended) IORB; 1 = IORB extended to at least 6+2*$AF items. Set by user. (See I_EXT below.)
		C-F	Function code. Driver or LPH function, see Table 6-1.
2+$AF	I_ADR	0-31	Buffer address. 2-word pointer.
2+2*$AF	I_RNG	0-15	Range. Number of bytes to be transferred. Used as input field for cartridge disk or mass storage unit.
3+2*$AF	I_DVS	0-15	Device-specific information.
4+2*$AF	I_RSR	0-15	Residual range. Indicates the number of bytes <u>not</u> transferred. Filled in by the system on completion of the order. Used by the cartridge disk and mass storage unit drivers as a data offset value.
5+2*$AF	I_ST	0-15	Modified device status. Shows mapping of hardware status into software status format. See Table 6-4. Set by user as input field high-order bits of sector number of mass storage unit. Set by system after I/O completion.
6+2*$AF	I_EXT	0-7	Left byte. Number of words, in binary, in the IORB extension, not including this I_EXT word.
		8-15	Right byte. Number of words, in binary, in physical I/O part of IORB extension, not including this I_EXT word. This count must be less than or equal to the total extension length specified in the left byte (0-7). This word is present only when the B(E) bit in I_CT2 is 1. (See Section 7 for a description of IORB extensions.)

Table D-7. Summary of IORB Fields for Operator Interface

Word	Label	Bit(s)	Contents
$AF	I_CT1	9(W)	For a $OPMSG call, the setting of the W-bit in the output IORB controls return to the caller. For a $OPRSP call, the setting of the W-bit in the input IORB controls return to the caller; the setting of the W-bit in the output IORB has no significance. For either call, return to the caller is immediate if the significant W-bit is on. If the significant W-bit is off, return to the caller occurs after the order is completed.
1+$AF	I_CT2	0-7	LRN = 0.
		9(B)	Must be off if the input/output buffer begins at the left byte of the word whose address is contained in word 3 (I_ADR) of this IORB. Must be on if the input/output buffer begins at the right byte.
2+$AF	I_ADR	0-15	The word address of the message buffer (which contains an output message or is to receive an input message).
2+2*$AF	I_RNG	0-15	The buffer size in bytes. This is the length of an output message or the maximum length allowed for an input message.

SEMAPHORE REQUEST BLOCK FORMAT

Figure D-4 shows the format of the semaphore request block; Table D-8 shows its content.

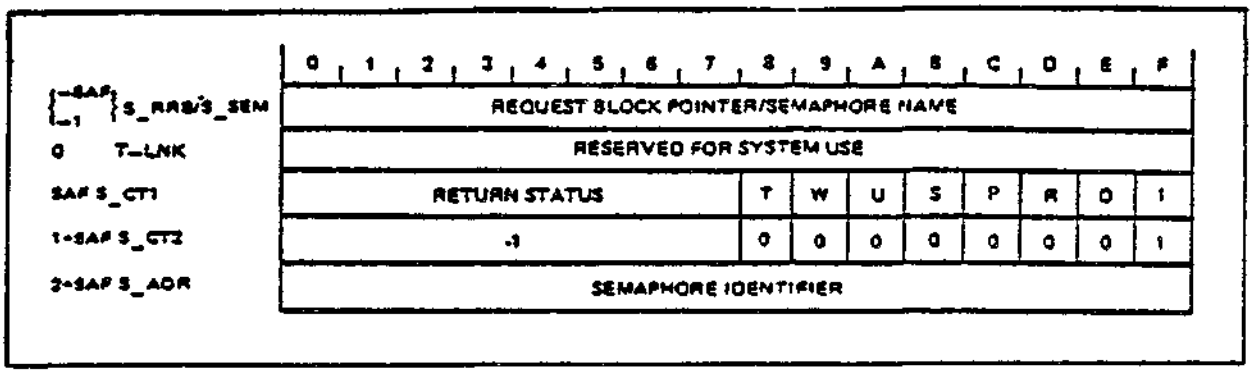


Figure D-4. Format of Semaphore Request Block

Table D-8. Contents of Semaphore Request Block

Word	Label	Bit(s)	Contents
-$AF -1	S_RRB S_SEM	0-31	Depending on the S- or R-bits of S_CTl, this field contains a 2-word task request block pointer (R-bit on), or a 1-word semaphore name (S-bit on). Set by user; used by system when request terminates.
0	S_LNK	0-15	Reserved for system use.
$AF	S_CTl	0-7	Return status
		8(T)	This bit is set (on) while the request using the block is executing; it is reset when the request terminates. The system controls this bit; user should not change it.
		9(W)	Wait bit. Set if the requesting task is <u>not</u> to be suspended pending the completion of the request that uses this block.
		A(U)	User bit. User may or may not use this bit; the system does not change it.
		B(S)	Release semaphore indicator. 0 = No release; 1 = Release, on completion, semaphore item named in S_SEM.

Table D-8 (cont). Contents of Semaphore Request Block

Word	Label	Bit(s)	Contents
$AF (cont)	S_CT1	C(P)	Must be set by user if SRB is to be referenced by a Wait Any ($WAITA) macro call. If set, SRB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D(R)	Return semaphore RB indicator. 0 = No dispatch; 1 = Dispatch task request block named in S_RRB after completion of this request.
		E(D)	Delete SRB indicator. Used usually with the B(S) and D(R) bits. 0 = No delete; 1 = Delete and, when task terminates, return memory to pool where SRB is first entry of its memory block.
		F	Implicit task start address. Must always be 1 for SRB.
1+$AF	S_CT2	0-7	Value is -1.
		8-14	Must be zero.
		15	Must be one.
2+$AF	S_ADR	0-15	Semaphore identifier - two ASCII characters.

TASK REQUEST BLOCK FORMAT

Figure D-5 shows the format of the task request block; Table D-9 shows its contents.

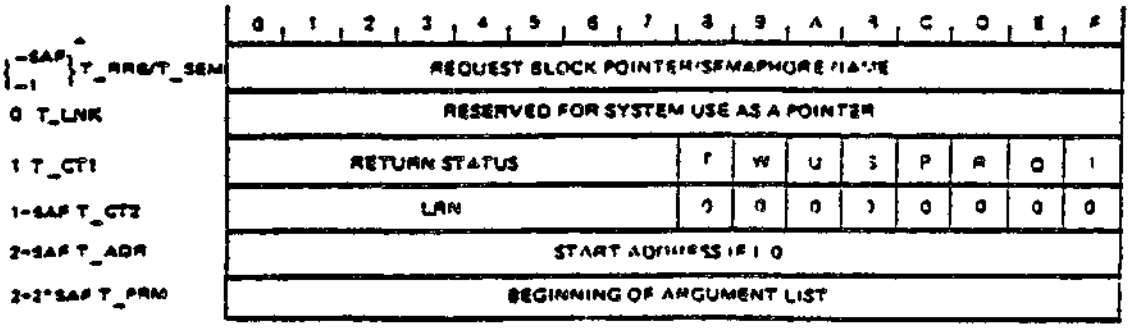


Figure D-5. Format of Task Request Block

Table D-9. Contents of Task Request Block

Word	Label	Bit(s)	Contents
-$AF -1	T_RRB/ T_SEM	0-31 0-15	Depending on the condition of the S- or R-bits of T_CT1, this field contains a 2-word task request block pointer (R-bit on), or a 1-word semaphore name (S-bit on). Set by user, used by system when request terminates.
0	T_LNK	0-31	Reserved for system use.
$AF	T_CT1	0-7	Return status.
		8(T)	This bit is set (on) while the request using this block is executing; it is reset when the request terminates. The system controls this bit; the user should not change it.
		9(W)	Wait bit. Set by user if the requesting task is <u>not</u> to be suspended pending the completion of the request that uses this block.
		A(U)	User bit. User may or may not use this bit; the system does not change it.

Table D-9 (cont). Contents of Task Request Block

Word	Label	Bit(s)	Contents
$AF (cont)	T_CT1 (cont)	B(S)	Release semaphore indicator. 0 = No release; 1 = Release, on completion, semaphore item named in T_SEM.
		C(P)	Must be set by user if TRB is to be referenced by a Wait Any ($WAITA) macro call. If set, TRB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D(R)	Return task RB indicator. 0 = No dispatch; 1 = Dispatch task request block named in T_RRB after completion of this request.
		E(D)	Delete TRB indicator. Used usually with the B(S) and D(R) bits. 0 = No delete; 1 = Delete and when task terminates, return memory to pool where TRB is first entry of its memory block.
		F	Implicit task start address. Must always be 1 for TRB.
1+$AF	T_CT2	0-7	Logical resource number (LRN).
		8-15	Must be zero.
2+$AF	T_ADR	0-15	Start address if the I-bit of T_CT1 is reset (zero).
2+2*$AF	T_PRM		Beginning of argument list.

PARAMETER BLOCK FORMAT

Figure D-6 shows the format of the parameter block.

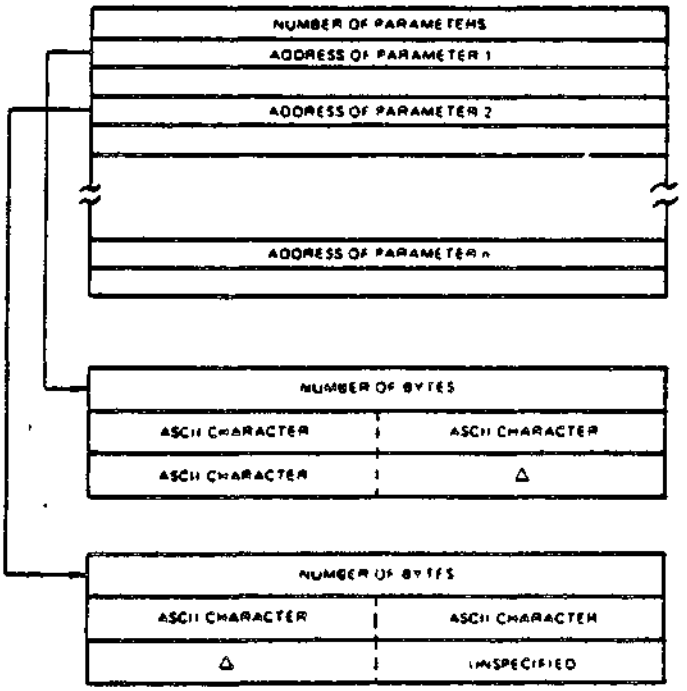


Figure D-6. Format of Parameter Block

NOTE

The parameter value strings need not be contiguous with the address portion of the parameter block; if the block is system-generated, each parameter will have a trailing blank that is not included in the byte count.

WAIT LIST FORMAT

Figure D-7 shows the format of the wait list.

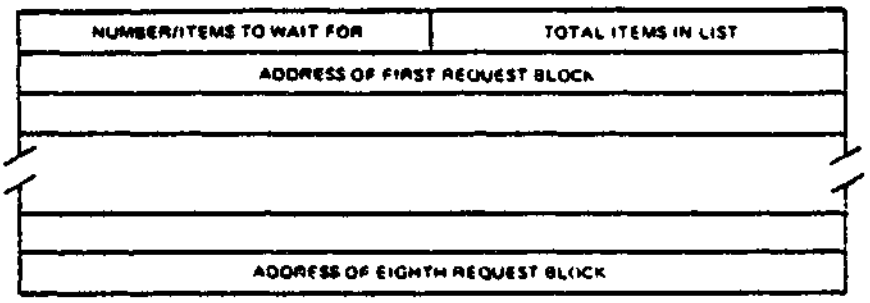


Figure D-7. Format of Wait List

MESSAGE GROUP REQUEST BLOCKS

Tables D-10, D-11, and D-12, respectively, show the content of the following message group request blocks:

- Message group control request block (MGCRB)
- Message group initialization request block (MGIRB)
- Message group recovery request block (MGRRB).

Templates for these request blocks are generated by the $MGCRT, $MGIRT, and $MGRRT macro calls, respectively.

The request blocks can be generated by the $MGCRB, $MGIRB, and $MGRRB macro calls, respectively.

Message group request blocks are used by the message facility for sending requests between task groups or tasks.

Table D-10. Message Group Control Request Block (MGCRB)

Word	Label	Bit(s)	Contents
0	MC_OS	0-31	Pointer; reserved for system use.
$AF	MC_MAJ		Major status.
		0-7	Reserved for system use.
		8(T)	This bit is set (on) while the request using this block is executing; it is reset when the request terminates. The system controls this bit; user should not change it.
$AF	MC_MAJ	9(W)	Wait bit. Set if the requesting task is not to be suspended pending the completion of the request that uses this block.
		A(U)	User bit. User may or may not use this bit; the system does not change it. Display processing uses this bit during a write.
		B(S)	Release semaphore indicator. Values: 0 = No release; 1 = Release (on closeout) of semaphore, which must be in MC_OS -1.
		C(P)	Must be set by user if MGCRB is to be referenced by a Wait Any ($WAITA) macro call. If set, MGCRB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D(R)	Return request block indicator. Values: 0 = No dispatch; 1 = Dispatch request block whose address must be contained in MC_OS -$AF, after closeout of this request.
		E(D)	Delete request block. Values: 0 = No delete; 1 = Delete, and return memory to the pool where MGCRB is the first entry of its memory block.
		F	I/O bit. Must be set.

Table D-10 (cont). Message Group Control Request Block (MGCRB)

Word	Label	Bit(s)	Contents
1+$AF	MC_OPT		General options:
		0-7	Reserved for system use.
		8	Must be 0.
		9	Byte index. 0 = Buffer begins in leftmost byte of the word; 1 = Buffer begins in rightmost byte.
		A	Must be 0.
		B	Must be 1 (extended MGCRB).
		C-F	Must be 0.
2+$AF	MC_BUF	0-31	Buffer pointer.
2+2*$AF	MC_BSZ	0-F	Buffer range (in bytes).
3+2*$AF	MC_DVS		Record-type code.
	MC_REC	0-F	On send, insert record-type code; on receive, return assigned record-type code.
4+2*$AF	MC_RSR	0-F	Residual range (in bytes).
5+2*$AF	MC_MRU	0-7	End message recovery unit (MRU). Reserved for system use.
	MC_WTI	8-F	Wait test indicator. 00 = Return null value to application; 01 = Wait.
6+2*$AF	MC_EXT		Extension mechanism.
		0-7	Binary value of 13+2*$AF, i.e., number of words in MGCRB following the extension word.
		8-F	Must be hexadecimal '7'.
7+2*$AF	Next 7 words		Reserved for system physical I/O use.

Table D-10 (cont). Message Group Control Request Block (MGCRB)

Word	Label	Bit(s)	Contents
14+2*$AF	MC_FNC	0-7	Function. Reserved for system use.
	MC_REV	8-F	Revision. Must be hexadecimal '2'.
15+2*$AF	MC_MGI		Message group id.
		0-F	Returned in the $MINIT and $MACPT macro calls.
16+2*$AF	MC_LVL		Enclosure level.
	MC_LVR	0-7	Enclosure level requested.
	MC_LVD	8-F	Enclosure level detected according to the following ASCII values: 0 = Not end of record; 1 = End of record; 2 = End of quarantine unit; 5 = End of message.
17+2*$AF	MC_PCI	0-F	Must be 0.
18+2*$AF	MC_VDP	0-31	Must be zero.
18+3*$AF	MC_TGI	0-F	Reserved for system use.
19+3*$AF	MC_TSK	0-31	Pointer. Reserved for system use.
19+4*$AF	MC_NPI	0-F	Must be 0.
22+3*$AF	MC_LEN	0-F	Length of text received.

Table D-11. Message Group Initialization Request Block (MGIRB)

Word	Label	Bit(s)	Contents
0	MI_OS	0-31	Pointer. Reserved for system use.

Table D-11 (cont). Message Group Initialization Request Block (MGIRB)

Word	Label	Bit(s)	Contents
$AF	MI_MAJ		Major status.
		0-7	Reserved for system use.
		8(T)	This bit is set (on) while the request using this block is executing; it is reset when the request terminates. The system controls this bit; user should not change it.
		9(W)	Wait bit. Set if the requesting task is not suspended pending the completion of the request that uses this block.
		A(U)	User bit. User may or may not use this bit; the system does not change it.
		B(S)	Release semaphore indicator. Values: 0 = No release; 1 = Release, on termination of this request, semaphore whose name must be in MI_OS -1.
		C(P)	Must be set by user if MGIRB is to be referenced by a Wait Any ($WAITA) macro call. If set, MGIRB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D(R)	Return request block indicator. Values: 0 = No dispatch. 1 = Dispatch,, after termination of this request, request block whose address must be contained in MI_OS -$AF.
		E(D)	Delete I/O request block. Values: 0 = No delete; 1 = Delete, and return memory to the pool where this MGIRB is the first entry of its memory block.
		F	I/O bit. Must be set.

Table D-11 (cont). Message Group Initialization Request Block (MGIRB)

Word	Label	Bit(s)	Contents
1+$AF	MI_OPT		General options.
		0-7	Reserved for system use.
		8-A	Must be 0.
		B	Must be 1 (extended MGIRB).
		C-F	Must be 0.
2+$AF	MI_BUF	0-31	Must be zero.
2+2*$AF	MI_BSZ	0-F	Buffer range (in bytes). Must be 0.
3+2*$AF	MI_MPD	0-F	Message path description identifier. Must be hexadecimal '01'.
4+2*$AF	MI_RSR	0-F	Residual range (in bytes).
5+2*$AF	MI_MDE	0-7	Must be 0.
	MI_IOP	8-F	Must be 0.
6+2*$AF	MI_EXT		Extension mechanism.
		0-7	Binary value of 31+2*$AF, i.e., number of words in MGIRB following the extension word.
		8-F	Must be hexadecimal '7'.
7+2*$AF	MI_DV2 (three words)	0-F 0-F 0-F	Maturity date/time in standard internal date/time format (see $INDTM).
14+2*$AF	MI_FNC	0-7	Function. Reserved for system use.
	MI_REV	8-F	Revision. Must be hexadecimal '2'.
15+2*$AF	MI_MGI		Message group id.
		0-F	Returned in the $MINIT and $MACPT macro calls.

Table D-11 (cont). Message Group Initialization Request Block (MGIRB)

Word	Label	Bit(s)	Contents
16+2*$AF	MI_PCM (Two words)	 0-F 0-F	 Must be 0. Must be 0.
18+2*$AF	MI_ADT	 0-7 8-F	Address type. Address type (initiator); must be hexadecimal '1'. Addr :s type (acceptor); must be hexa cimal '1'.
19+2*$AF	MI_NWI	0-F	Must be 0.
20+2*$AF	MI_NDI	0-F	Must be 0.
21+2*$AF	MI_MBI		Initiator mailbox name.
	(Six words)	0-F 0-F 0-F 0-F 0-F 0-F	Must be from 1 to 12 ASCII characters, blank-filled, left justified as specified when the mailbox was created, indicating that only messages with this identifier will be accepted; or six words of zeros, indicating that messages with any identifier will be accepted.
27+2*$AF	MI_NWA	0-F	Must be 0.
28+2*$AF	MI_NDA	0-F	Must be 0.
29+2*$AF	MI_MBA (Six words)	 0-F 0-F 0-F 0-F 0-F 0-F	Acceptor mailbox name. Must be from 1 to 12 ASCII characters specifying the acceptor mailbox id, blank-filled, left-justified.
36+2*$AF	MI_CNT	0-F	Count of number of active messages in the mailbox. Returned with $MCMG macro call.

Table D-11 (cont). Message Group Initialization Request Block (MGIRB)

Word	Label	Bit(s)	Contents
37+2*$AF	MI_TGI	0-F	Reserved for system use.
38+2*$AF	MI_TSK	0-31	Pointer. Reserved for system use.
38+3*$AF	MI_SIP	0-31	Security information pointer. Points to the security information block (SIB) that points to the logical submittor block containing the user id (SI_PER), the account id (SI_ACC), and the mode (SI_MOD).

Table D-12. Message Group Recovery Request Block (MGRRB)

Word	Label	Bit(s)	Contents
0	MR_OS	0-31	Pointer. Reserved for system use.
$AF	MR_MAJ		Major status.
		0-7	Reserved for system use.
		8(T)	This bit is set (on) while the request using this block is executing; it is reset when the request terminates. The system controls this bit; user should not change it.
		9(W)	Wait bit. Set if the requesting task is not to be suspended pending the completion of the request that uses this block.
		A(U)	User bit. User may or may not use this bit; the system does not change it.

Table D-12 (cont). Message Group Recovery Request Block (MGRRB)

Word	Label	Bit(s)	Contents
$AF (cont)	MR_MAJ (cont)	B(S)	Release semaphore indicator. Values: 0 = No release; 1 = Release, on closeout, of semaphore which must be in MC_OS -1.
		C(P)	Must be set by user if MGRRB is to be referenced by a Wait Any ($WAITA) macro call. If set, MGRRB can be referenced only by $WAIT or $WAITA issued by the requesting task.
		D(R)	Return request block indicator. Values: 0 = No dispatch; 1 = Dispatch request block, whose address must be in MC_OS -$AF, after closeout of this request.
		E(D)	Delete I/O request block. Values: 0 = No delete; 1 = Delete, and return memory to the pool where MGRRB is the first entry of its memory block.
		F	I/O bit. Must be set.
1+$AF	MR_OPT		General options.
		0-7	Reserved for system use.
		8-A	Must be 0.
		B	Must be 1 (extended MGRRB).
		C-F	Must be 0.
2+$AF	MR_BUF	0-31	Pointer. Must be 0.
2+2*$AF	MR_BSZ	0-F	Buffer range. Must be 0.
3+2*$AF	MR_ITP	0-F	Must be 0.
4+2*$AF	MR_RES	0-F	Residual range. Reserved for system use.

Table D-12 (cont). Message Group Recovery Request Block (MGRRB)

Word	Label	Bit(s)	Contents
5+2*$AF	MR_RSN	0-7	Reason-for-terminate code. 0 = Normal message group termination; 22-26 = User-defined abnormal termination of message group.
		8-F	Reserved for system use.
14+2*$AF	MR_FNC	0-7	Function. Reserved for system use.
	MR_REV	8-F	Revision. Must be hexadecimal '02'.
15+2*$AF	MR_MGI	0-F	Message group id. Returned in the $MINIT and $MACPT macro calls.
17+2*$AF	MR_CNC	0-F	Reserved for system use.
16+2*$AF	MR_FMT	0-31	Pointer. Must be 0.
18+3*$AF	MR_MRU (Two words)	0-F	Reserved for system use.
		0-F	Reserved for system use.
19+3*$AF	MR_AMU (Two words)	0-F	Reserved for system use.
		0-F	Reserved for system use.

Appendix E
BACKUP AND RECOVERY

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 allows you 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 two 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 two hours, the memory image is destroyed and the power resumption facility disabled. When power is restored, the user 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. File recovery provides file integrity in the event of a system failure.

File recovery is provided through three distinct functions:

- "Before image" recording, which preserves a record prior to its being updated.
- "Cleanpoint" or "checkpoint" declarations, which are issued in your program and define a point at which all updates are complete. When the updates are complete, the associated before images are destroyed.
- "Rollback" or "restart" functions, which return the files to their unaltered state by applying all before images that have been recorded since the last cleanpoint.

The cleanpoint and rollback 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 the 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 the program when you reach a point in processing where 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 programs 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 Restore.

The Restore program reconstructs the file structures copied by the Save program. If a file being restored already exists on the volume (or volumes), the Restore program replaces the current file contents with the file data saved by the Save program. (The access list is not altered.) If a file being restored does not exist on the volume, the Restore program 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 two hours.

(It cannot, however, preserve the state of the I/O controllers or ensure that no operational changes have been made to the mounted volumes.)

If fewer than two 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 device (ATD) line protocol handler or the teleprinter (TTY) line protocol handler (see the System Building manual for information on configuring devices serviced by these LPHs; see Sections 8 and 12 of this manual for information about the ATD and TTY LPHs, respectively).
- Restart application tasks that were active at the time of the failure.

Implementing the Power Resumption Facility

The power resumption facility must be included in the MOD 400 Executive at system building. The central processor must contain a memory save and autorestart unit that has been activated by the operator (see the System User's Guide for activation procedures).

When power resumption is specified in the system building dialog, 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 ATD/TTY-associated device is not to be automatically reconnected, you must edit the CLM file to remove the -RECONNECT argument from the STTY directive generated for the device.

Power Resumption Procedures

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/TTY-associated communication devices that were online at the time of the failure, except for those devices designated as not reconnectable.

- Restarts 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 (if any).

- Reestablishes the integrity of mounted volumes.

- Restarts application tasks that were active when the power failure occurred. (For an application to successfully complete after being restarted, it may have to contain user-written code to handle power failure/power resumption.)

In order for an application task to be notified when a power resumption has occurred, it must connect its own trap handler and enable trap 53. Trap 53 condition will be signaled when the task becomes active and is issuing its own instructions (not executing Executive instructions). See "Trap Handling" in Appendix A.

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

FILE RECOVERY

File recovery 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 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 the data files, the data files are rolled back to a previously established state.

Designating Recoverable Files

File recovery is optional. You can 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 (or task group 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.

If the tasks in a task group have only sharable files, only one recovery file exists for the group. If any task in a task group has an exclusive file, one recovery file is created for each task in the group.

All recovery files are created subordinate to your working directory. 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 - Group identifier
tt - 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 can 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 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, you 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).

Note that the file system 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, you request a rollback by coding a CALL "ZCROLL" statement. If programming in Assembly language, you request a rollback by coding a $ROLBK macro call. When a rollback is requested, 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

When the system is reinitialized following a system failure, it checks for the existence of recovery files. If recovery files do not exist, files had not previously been declared as recoverable or updates had not previously been made to recoverable files. If recovery files do exist, the system failure occurred while updates were being made to a file that had the recover attribute. If recovery files exist, the operator should issue the Recover command so that the system will perform a rollback of all recoverable data files. See the System User's Guide for details.

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 your memory and the current state of tasks, files, and screen forms onto a checkpoint file previously assigned. The system then takes a cleanpoint so that recoverable files are synchronized with that checkpoint. See "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 are 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 ECs can be checkpointed.)

Checkpoint File Assignment

You can 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 have specified.

When designating the checkpoint file, you 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, you 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, you 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 cheekpointable 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 which 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 of nonsharable bound units.

3. Memory locations and pathnames of sharable bound units.

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

5. Status and position of all active 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 restart.
- 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 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 or must be in a swap pool. 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), you 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 will be aborted. Specification of the -WTMEM argument of the restart command will cause 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 will cause 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.

RESTART PROCESSING

When the Restart command is issued, 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; see "Requesting a Restart" above.

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 the 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.

Appendix F
ASCII AND EBCDIC CHARACTER SETS

Tables F-1 and F-2 illustrate the ASCII and EBCDIC character sets, respectively. In addition to the ASCII characters, Table F-1 shows the hexadecimal equivalents; Table F-2 shows the binary and hexadecimal equivalents of the EBCDIC character set.

Following are lists of the control characters and special graphic characters that appear in the two tables:

CONTROL CHARACTERS

ACK	Acknowledge	GE	Graphic Escape
BEL	Bell	GS	Group Separator
BS	Backspace	HT	Horizontal Tab
BYP	Bypass	IFS	Interchange File Separator
CAN	Cancel	IGS	Interchange Group Separator
CC	Cursor Control	IL	Idle
CR	Carriage Return	IRS	Interchange Record Separator
CU1	Customer Use 1	IUS	Interchange Unit Separator
CU2	Customer Use 2	LC	Lowercase
CU3	Customer Use 3	LF	Line Feed
DC1	Device Control 1	NAK	Negative Acknowledgement
DC2	Device Control 2	NL	New Line
DC3	Device Control 3	NUL	Null
DC4	Device Control 4	PF	Punch Off
DEL	Delete	PN	Punch On
DLE	Data Link Escape	RES	Restore
DS	Digit Select	RLF	Reverse Line Feed
EM	End of Medium	RS	Reader Stop
ENQ	Enquiry	SI	Shift In
EO	Eight Ones	SM	Set Mode
EOT	End of Transmission	SMM	Start of Manual Message
ESC	Escape	SO	Shift Out
ETB	End of Transmission Block	SOH	Start of Heading
ETX	End of Text	SOS	Start of Significance
FF	Form Feed	SP	Space
FS	Field Separator	STX	Start of Text
SUB	Substitute	UC	Uppercase
SYN	Synchronous Idle	US	Unit Separator
TM	Tape Mark	VT	Vertical Tab

SPECIAL GRAPHIC CHARACTERS

¢	Cent Sign	>	Greater-than Sign
.	Period, Decimal Point	?	Question Mark
<	Less-than Sign	`	Grave Accent
(	Left Parenthesis	:	Colon
+	Plus Sign	#	Number Sign
¦	Logical OR	@	At Sign
&	Ampersand	'	Prime, Apostrophe
!	Exclamation Point	=	Equal Sign
$	Dollar Sign	"	Quotation Mark
*	Asterisk	~	Tilde
)	Right Parenthesis	{	Opening Brace
;	Semicolon	⌐	Hook
¬	Logical NOT	┬	Fork
-	Minus Sign	}	Closing Brace
/	Slash	\	Reverse Slant
|	Vertical Line	├	Chair
,	Comma	|	Long Vertical Mark
%	Percent	[	Opening Bracket
_	Underscore	]	Closing Bracket
^	Circumflex		

Table F-1. ASCII/Hexadecimal Equivalents

	H1							
H2	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table F-2. EBCDIC/Hexadecimal/Binary Equivalents

TABLE C 2 EBCDIC/HEXADECIMAL/BINARY EQUIVALENTS

BIT POSITIONS 4, 5, 6, 7	SECOND HEXADECIMAL DIGIT	00				01				10				11			
		00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	NUL	DLE	DS		SP	&							{[a]	}[a]	\[a]	0
0001	1	SOH	OC1	SOS				/		a	j	~[a]		A	J		1
0010	2	STX	OC2	FS	SYN					b	k	s		B	K	S	2
0011	3	ETX	TM							c	l	t		C	L	T	3
0100	4	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	5	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	6	LC	BS	ETB	UC					f	o	w		F	O	W	6
0111	7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
1000	8	GE[a]	CAN							h	q	y		H	Q	Y	8
1001	9	RLF[a]	EM						\[a]	i	r	z		I	R	Z	9
1010	A	SMM	CC	SM		¢	!	.[a]	:								![a]
1011	B	VT	CU1[a]	CU2[a]	CU3[a]		$	,	#								
1100	C	FF	IFS		DC4	<	*	%	@					S[a]		d[a]	
1101	D	CR	IGS	ENQ	NAK	(	)	_									
1110	E	SO	IRS	ACK		+	;		=					ψ[a]			
1111	F	SI	IUS	BEL	SUB	|	¬	>	"								EO[a]

BIT POSITIONS 0, 1

BIT POSITIONS 2, 3

FIRST HEXADECIMAL DIGIT

[a]THIS CHARACTER IS NOT SUPPORTED IN THE 2780 CHARACTER SET

Appendix G
DEVICE-SPECIFIC CONTROL CHARACTERS

This appendix lists the nonalphanumeric control characters for devices supported by the communications subsystem.

NOTE

In this appendix, a slash between two characters indicates that both keys are pressed simultaneously, e.g., CTRL/H indicates that the CTRL key and H key are pressed at the same time.

Table G-1. TTY Nonalphanumeric Control Characters

Character	Hexadecimal Value	Function	Key Strokes
ENQ	05	Answer back	CTRL/E
BEL	07	Ring Bell	CTRL/G
BS	08	Backspace (nondestructive cursor backward)	CTRL/H

Table G-1 (cont). TTY Nonalphanumeric Control Characters

Character	Hexadecimal Value	Function	Key Strokes
LF	0A	Line feed	CTRL/J
FF	0C	Form feed (clear screen)	CTRL/L
CR	0D	Carriage return	CTRL/M
SP	20	Space	CTRL/P or space bar

NOTE

In a terminal with lowercase capability, uppercase characters require the use of the shift.

Table G-2. VIP Nonalphanumeric Control Characters

Character	Hex Value	Function	Key Strokes
BS	08	Backspace.	CTRL/H
HT	09	Horizontal tab.	CTRL/I
LF	0A	Line feed.	CTRL/J or LINE FEED
FF	0C	Form feed.	CTRL/L
CR	0D	Carriage return.	CTRL/M or RETURN
ESC	1B	First character of a 2-, 3-, or 4-character escape sequence used for VIP terminal control.	ESC
SP	20	Space.	CTRL/P or space bar

Appendix H
SUBSYSTEM MODULES

This appendix describes subsystem modules: their purpose, structure, and interface with the Edit Profile and List Profile utilities. The intent of this appendix is to help developers create their subsystem modules according to the requirements of these two utilities. For the sake of clarity, references to the profiles file will be more technical than in the user-oriented manuals. Specifically, a 'section' of a user profile is a record in the profiles file; the 'attributes' of a section are the fields of the record; a 'section id' is a two-character record-type identifier located in the record header.

SUBSYSTEM RECORDS

A subsystem record is 188 bytes long and consists of a system-defined portion and a subsystem-defined portion. The system-defined portion extends from bytes 0 through 59. This area of the record cannot be accessed by the subsystem or subsystem module.

The remainder of the record, defined by the subsystem, is subdivided into two regions. Subsystem region 1, also known as the access level region, extends from bytes 60 through 97. This region can be read by the subsystem proper but can be written to only by Edit Profile (via a subsystem module).

Subsystem region 2 extends from bytes 98 through 188. It can be read or written to both by the subsystem proper and Edit/List Profile (via a subsystem module). The subsystem proper uses profiles file macro calls, documented in Volume II of this manual, to read and write the subsystem records.

Subsystem-defined fields processed by Edit/List Profile (via a subsystem module) must begin on an even byte (word boundary).

EDIT PROFILE (EP) SUBSYSTEM MODULES

When the System Administrator uses Edit Profile's ADD, MOD, or STATS functions on a subsystem record, EP calls the subsystem module of that record type. The module provides EP with code and data needed to perform the functions.

The module is a separate bound unit, linked non-sharable, and without overlays. The naming convention is EP_id, where id is the two-character record type identifier (section_id). The module resides in a directory under the loader's search rules.

Edit Profile (EP) Module Contents

An EP module contains up to seven fundamental elements listed below. The location within the module of any of these elements is not important except for the pointer array, which must begin at word one. Word zero (start address) must be the instruction jmp $B5 (8385). This prevents the module from being executed as an ECL command, which would cause a trap.

Elements of an EP module are:

1. Pointer array.
2. MOD function message number.
3. MODIFY routine.
4. Default values.
5. ADD routine.
6. Stat-names message number.
7. Table of statistics field descriptors.

Elements 6 and 7 need only exist if the subsystem record contains statistics fields for display by the STATS function.

POINTER ARRAY

The pointer array starts at word 1 of the subsystem module and contains six IMA pointers to the elements (2 through 7) listed above (in the same order).

If the subsystem record contains no statistics, then the pointers to elements 6 and 7 are zero.

The PTRAY assembler control statement is useful for creating the pointer array.

MOD FUNCTION MESSAGE NUMBER

This is a five digit number defined as a hex string constant (i.e., DC Z'nnnnn'), identifying a message in the message library.

The message is actually a table of names that is displayed in list form by Edit Profile under the MOD function.

The entries in the table tell the user which fields can be modified in the record. You may choose to specify individual field names or, instead, group fields into categories, in which case the table would contain the category names.

The latter may be more helpful to the user but would entail more work for the subsystem module developer because Edit Profile processes only the initial table entries; your MOD routine would have to display the elements of the chosen category. In either case, the MOD routine must be coordinated with the make-up of this table.

The format of the table is as follows:

```
name 1/name 2/name 3...name n!
```

Note that the entries are lower case, the slash character is used as a separator, and the exclamation point is the end-of-table marker. All other characters, including space, are legal for a field name. The maximum length of an individual entry is 22 characters. The maximum total length is 240 characters (this is a message library limitation).

The following example shows the first pointer in the array pointing at the message number. Also shown is the message as it would appear in the message library.

```
ptrary DC <msgnum
       .
       .
       .
       .

msgnum DC Z'26301'
```

Message library entry:

```
263010100000 login_id/login line defaults/current
terminal/language key/login traits/password status!
```

NOTE

Creation of the message and insertion into the message library must follow message reporter specifications, which are found in System Messages (CZ16).

Under the MOD function, Edit Profile retrieves the table from the message library and displays it as a list with each entry assigned an incremental number. Edit Profile also displays a 'NONE OF THE ABOVE' option following the last table entry, and then prompts the user's selection. Figure H-1 shows the format of this list.

```
XX Section Menu

(1)    name 1
(2)    name 2
(3)    name 3
         .
         .
         .
(n)    name n
(n+1)  NONE OF THE ABOVE

Selection:
```

Figure H-1. MOD Function List Format

MODIFY ROUTINE

The subsystem module must contain a routine to change in memory the field that the user has selected.

When the MOD function is executed, Edit Profile does the following:

1. Loads ($BULD) the appropriate subsystem module into memory.

2. Reads the subsystem record from the profiles file into Edit Profile's memory buffer.

3. a. Displays in menu-like form the subsystem-supplied table of names, each name assigned a number from 1 through n.

   b. Prompts the user's selection.

4. Validates the selection to be within the range 1 through n.

5. Converts the selected number to hexidecimal and loads it into $R1 (e.g., if 10 is selected, $R1 = 000A).

6. Points $B1 at the access level region (byte 60) of the record in memory.

7. Does a link and jump $B5 to the subsystem modify routine.

The subsystem MOD routine uses the value in $R1 to determine which entry was selected from the list.

If a category was selected (assuming one was offered), the routine displays a similar type of list containing the entries under the chosen category, and then prompts for the user's selection.

When the user's selection identifies a specific field, the MOD routine prompts the user to supply the field's new contents/value.

The routine replaces (in memory) the old field contents with the new, puts zero in $R1 to signify a clean return, and jumps back to Edit Profile at the address originally in $B5.

NOTE

The MOD routine can alter only bytes 60 through 188 of the subsystem record (counting from physical byte 1).

The subsystem MOD routine:

- Uses the error-out and user-in paths for all dialogue
- Does input verification on user responses
- Does its own error reporting
- Accepts YES, Y, NO and N as responses to yes/no questions
- Recognizes '?' as the help key and responds with a help message if possible, or displays 'No help available'
- Recognizes '<' as a 'back-up' key and returns to the previous prompt/question. If '<' is received on the first prompt, puts -1 in $R1 and returns to Edit Profile.

When the subsystem MOD routine returns, Edit Profile checks $R1 and takes one of the following actions:

- $R1 = 0 (normal return). EP picks up at step 3 redisplaying the list but now offering two action keys (A) Accept and (N) Negate. If 'A' is selected, the record in memory (containing the changes) is written to the profiles file and the MOD function is exited.

- If 'N' is selected, the original record is read from the profiles file into memory, overwriting the changes made by the subsystem MOD routine. EP then announces that the changes have been negated and picks up again at step 3.

- $R1 = -1 (back-up key used). No change was made; EP redisplays the list (step 3). The action keys are not offered.

- $R1 ≠ 0 or -1 (abnormal return). EP displays an error message and exits the MOD function.

SUBSYSTEM DEFAULT VALUES

The subsystem default values are a total of 128 bytes long and represent the initial contents of the subsystem-defined portion of the record (bytes 60 through 188).

When an ADD function is entered, EP creates a skeleton record by copying the values into a memory bufffer before activating the subsystem ADD routine.

ADD ROUTINE

The subsystem module must contain a routine to build a new subsystem record in memory field-by-field using information gained from an interactive dialog with the system administrator.

When an ADD function is entered, Edit Profile builds a skeleton record in memory containing the subsystem default values. Edit Profile then sets $B1 pointing at the subsystem defined portion (byte 60) of the skeleton and does a link and jump $B5 to the subsystem ADD code.

The subsystem code issues directives and questions through the error-out file requesting the administrator to respond with field contents or answers to subsystem-specific questions. The responses are accepted through user-in and validated. If the response is invalid, the code issues a message to error-out indicating the problem and then re-issues the directive or question. Valid responses are put into the skeleton record at the appropriate offsets.

As in the MOD routine, the ADD routine must recognize the '?' (help) and '<' (back-up) keys as defined by Edit Profile.

Return to EP is at the orginal $B5 address. Upon return, EP checks the contents of $R1 as it does upon return from the MOD routine.

STAT-NAMES MESSAGE NUMBER

This is a 5-digit number defined as a hex string constant identifying a message in the message library. The message is a table of the names of the statistics fields that the STATS function is to display.

The format of the table is exactly the same as the one displayed by the MOD function and already described under "MOD Function Message Number."

STATS DESCRIPTOR TABLE

The stats descriptor table contains a 3-word entry for each statistic field in the record. The three words define the stat-type, offset, and size of the field.

The stat-type word is a number indicating one of the stat-types shown in Table H-1.

Table H-1. Edit Profile Statistic Field Types

Type	Number
Decimal count	1
Hex count	2
Elapsed internal time	3
Internal date/time	4

The offset word indicates the offset of the field in words. (Keep in mind that the offset of the sixtieth word, for example, is 59.)

The size word indicates the length of the field in words.

The following example shows a typical stats descriptor table.

```
stats  DC 3, 49, 3 (elapsed time; offset: 49; size: 3 words)
          1. 52, 1 (decimal count; offset: 52; size: 1 word)
          2, 53, 2 (hex count; offset: 53; size: 2 words)
          4, 55, 3 (internal date/time; offset: 55; size: 3 words)
```

LIST PROFILE (LP) SUBSYSTEM MODULES

When List Profile processes a subsystem record, it calls the LP subsystem module of that record type. The module is a separate bound unit, linked non-sharable and without overlays. The naming convention is LP_id, where id is the two character record type identifier (section_id). The module resides in a directory under the loader's search rules.

LP Module Contents

An LP subsystem module contains up to four fundamental elements listed below. The location within the module of any element is not important except for the pointer array, which must begin at word one. Word zero (start address) must be the instruction jmp $B5 (8385). This prevents the module from being executed as an ECL command, which would cause a trap.

Elements of an LP subsystem module are:

1. Pointer array.
2. Message number.
3. Descriptor table.
4. Special-field routine.

Element 4 need exist only if the subsystem record contains any special fields (see "Special Field Routine" later in this appendix).

POINTER ARRAY

The pointer array starts at word one of the module and contains three IMA pointers to the elements 2, 3, and 4 listed above (in the same order).

If the module does not contain a special-field routine, then the associated pointer is null (zeros).

Message Number

This is a five digit number defined as a hex string constant (i.e., DC Z'nnnnn'), identifying a message in the message library.

The message is actually a table of the field names in the subsystem record.

The field names are displayed in a column by List Profile when the record is listed.

In format, the table is exactly the same as Edit Profile's table of modifiable field names, described earlier in this appendix under "MOD Function Message Number."

Creation of the message and insertion into the message library must follow the rules of the Message Reporter, which are explained in System Messages (CZ16).

DESCRIPTOR TABLE

The descriptor table contains a three-word entry for every field in the record.*

The three words define: field type, offset, and size.

The field type word is a number indicating one or more field types shown in table H-2.

Table H-2. List Profile Field Types

Type	Number
Decimal count	1
Hexadecimal count	2
Elapsed internal time	3
Internal date/time	4
RFU	5
Special field	6
ASCII	7
Radix 40	8
Bit string	9

The offset word indicates the offset of the field in words.

The size word indicates the length of the field in words.

*The table of field names and the descriptor table should contain entries only for subsystem-defined regions (i.e., bytes 60 through 188). The table should not contain entries for fields in the system-defined region of the record. No entry need exist for any unused area(s) in bytes 60 through 188.

SPECIAL-FIELD ROUTINE

Any field in the record whose contents require interpretation by the subsystem, such as an indicator word, cannot be processed by List Profile alone. Such fields require the existence of a subsystem module routine to interpret and display the field contents.

When List Profile encounters a special field, it builds an argument structure comprising the following elements:

- Address of List Profile's memory buffer containing the subsystem record
- Address of List Profile's output buffer (points at the control word)
- Offset of the current field (in words)
- Length of the current field (in words)
- Byte offset into List Profile's output buffer for placement of field contents.

The addresses are two words in length; the last three arguments are one word each.

List Profile places the ASCII field name in its output buffer, points $B4 at the argument structure, and does a link and jump $B5 to the subsystem routine.

The routine uses the information in the argument block to do the following:

- Determine which special field is being processed (if there is more than one in the record)
- Interpret the contents of the field
- Place the translated ASCII meaning in List Profile's output buffer at the supplied offset
- Display the output buffer to user-out.*

The routine maintains control for as long as it needs. In the case of an indicator word, for example, each bit may require a separate display, in which case the routine would have to clean the buffer after a $USOUT, interpret and print the next bit, and so on.

---

*List Profile's output buffer is 80 bytes long including the control word. The maximum length of a field's contents is 50 ASCII characters.

When done, the routine uses $R1 as a status register: a zero in $R1 signifies normal return; a non-zero value in $R1 signifies an error. The subsystem returns to List Profile at the address originally supplied in $B5.

ASCII-ONLY SUBSYSTEM RECORDS

The Edit Profile and List Profile utilities can add, modify, and list a subsystem record without the use of a subsystem module if both of the following conditions are met:

- The subsystem-defined portion of the record contains only ASCII data
- The subsystem was declared (by the DEC function) to operate in this mode of record maintenance.

Edit Profile and List Profile view an ASCII-only record as having two data regions: region one in bytes 60 through 97; region two in bytes 98 through 188.

Under the ADD and MOD functions, Edit Profile prompts the System Administrator as follows:

```
Enter data for region 1:

Enter data for region 2:
```

Any unused portion of either region is blank filled. List Profile displays all of region 1 (38-characters) and 50 of the 90-characters in region 2.