Honeywell

LEVEL 68

SOFTWARE

MULTICS BULK INPUT/OUTPUT

Roach

MULTICS BULK INPUT/OUTPUT ADDENDUM D

SUBJECT

Additions and Changes to the Manual

SPECIAL INSTRUCTIONS

This is the fourth addendum to CC34, Revision 1, dated March 1979.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Throughout the manual, change bars in the margin indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of this manual.

Note:
Insert this cover after the manual cover to indicate the updating of the document with Addendum D.

SOFTWARE SUPPORTED

Multics Software Release 10.0

ORDER NUMBER

CC34-01D

July 1982

34755
7.5C682
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

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

Remove	Insert
title page, preface	title page, preface
iii through vi	iii through vi
1-1, 1-2	1-1, 1-2
2-7, 2-8	2-7, 2-8
2-21, 2-22 2-22.1, blank	2-21, 2-22 2-22.1, blank
3-21, 3-22	3-21, 3-22 3-22.1, blank
3-35 through 3-38	3-35 through 3-38
3-51 through 3-54	3-51 through 3-55, blank
	A-4.1, blank
C-1, C-2	C-1, C-2
G-3, G-4	G-3, G-4 G-4.1, blank
i-1 through i-4	i-1 through i-4

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

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

© Honeywell Information Systems Inc., 1982

File No.: 1L13, 1U13

7/82

CC34D

SERIES 60 (LEVEL 68)

MULTICS BULK INPUT/OUTPUT

SUBJECT

Information Needed by System Administrators and Operators in the Management of Bulk Input/Output

SPECIAL INSTRUCTIONS

This manual supersedes the preliminary edition of *Multics Bulk Input/Output*, Order No. CC34, Revision 0. Each section has been extensively revised; change bars in the margins indicate technical changes and additions, and asterisks indicate deletions.

Much of the information in this manual on the administration and operation of the I/O daemon supersedes material that has been available in the *Multics Operators' Handbook*, Order No. AM81 and the *Multics Administrators' Manual – System*, Order No. AK50. The superseded material has been removed from these manuals.

Information about bulk input/output needed by users is in the *Multics Programmers' Manual – Communications Input/Output*, Order No. CC92.

Please refer to the Preface for more specific information concerning changes to this manual.

SOFTWARE SUPPORTED

Multics Software Release 7.0

ORDER NUMBER

CC34, Rev. 1

March 1979

Honeywell

PREFACE

The Multics Bulk Input/Output manual contains information needed by system administrators and operators in the management of the daemons that handle input/output to unit record devices (card readers, printers, and punches).

Other manuals that provide additional information and that are referenced in this manual include:

Document	Referred to in Text As
Multics Programmers' Manual (MPM) Reference Guide (Order No. AG91)	MPM Reference Guide
MPM Commands and Active Functions (Order No. AG92)	MPM Commands
Multics Operators' Handbook (Order No. AM81)	MOH
Multics Administrators' Manual (MAM) System (Order No. AK50)	MAM System

In addition to the above documents, the reader is referred to the appropriate peripheral equipment manuals for information on particular devices.

Changes in CC34, Revision 1, Addendum D

Changes have been made in Sections 2 and 3 and in Appendixes A, C, and G. The iod_command command in Appendix A is new; therefore, it does not contain change bars.

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

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

© Honeywell Information Systems Inc., 1982

File No.: 1L13, 1U13

7/82

CONTENTS

Page

Section 1 Introduction . . . 1-1

Section 2 Directory Structure and Data Bases of the I/O Daemon . . . 2-1
- I/O Daemon Directories . . . 2-1
  - Contents of daemon_dir_dir Directory . . . 2-1
  - Contents of io_daemon_dir Directory . . . 2-1
  - Contents of cards Directory . . . 2-3
- I/O Daemon Tables . . . 2-3
  - I/O Daemon Tables Source Language . . . 2-3
    - Syntax . . . 2-3
    - Statements . . . 2-4
    - Substatements for Lines . . . 2-4
    - Substatements for Devices . . . 2-5
    - Substatements for Request Types . . . 2-6
    - Source File Example . . . 2-7
    - Major and Minor Devices . . . 2-8
    - Substatements for Minor Devices . . . 2-9
    - Source File Example Using Minor Devices . . . 2-9
    - AIM Features . . . 2-10
    - Substatements for Device Classes . . . 2-11
    - Substatement for Default Request Type 2-12
    - Source File Example Using AIM . . . 2-12
  - Standard Driver Modules . . . 2-13
    - printer_driver_ MODULE . . . 2-14
    - punch_driver_ MODULE . . . 2-14
    - reader_driver_ MODULE . . . 2-14
    - spool_driver_ MODULE . . . 2-14.1
    - remote_driver_ MODULE . . . 2-15
      - Normal setup of the remote_driver_ (Type I stations) . . . 2-15
      - Setup for stations that cannot input commands (Type II stations) 2-17
      - Remote Driver <string> Arguments . 2-17
- Creation and Maintenance of I/O Daemon Tables . . . 2-18
- Creation and Maintenance of I/O Daemon Queues . . . 2-19
- Maintenance of AIM Features . . . 2-20
- Request Type Info Segments . . . 2-20
  - Syntax For The Request Type Info Source Segment . . . 2-21
  - Example of a Request Type Info Source Segment . . . 2-24

Section 3 Operation of the I/O Daemon . . . 3-1
- Login and Initialization of the I/O Coordinator . . . 3-1
- Communicating with the Coordinator . . . 3-2
- Interrupting the Coordinator . . . 3-2
- Coordinator Commands . . . 3-3
  - list . . . 3-3
  - print_devices . . . 3-3
  - wait_status . . . 3-3
  - term . . . 3-4
  - restart_status . . . 3-4
- Login and Initialization of Device Drivers . 3-4

Page

- Logging in a Driver 3-4
- Driver Initialization 3-4
- Driver Initialization When Using Device Classes 3-7
- Terminals That Control The Driver 3-8
  - Master Versus Slave Functions 3-8
  - Driver Initialization With A Control Terminal 3-8
- Driver Command Levels 3-9
  - Normal Driver Command Level 3-10
  - Request Command Level 3-10
  - Quit Command Level 3-10
- Standard Driver Commands 3-10
  - General Control Commands 3-11
  - Control Commands After Interrupting a Request 3-11
  - Information Commands 3-12
  - Coordinator Communication Commands . . . 3-12
  - Commands For Terminal Control 3-12
  - Error Recovery Commands 3-12
- Device Specific Driver Commands 3-13
- Making The Driver Ask For A Command 3-14
- Entering Commands From A Multifunction Device Card Reader 3-14
- Using Preprinted Accountability Forms On The Control Terminal 3-14
- Device Specific Driver Operations 3-16
  - Operation of the Printer Driver 3-16
    - Login and Initialization 3-16
    - Limitations 3-17
    - Processing Requests 3-17
  - Operation of the Punch Driver 3-18
  - Operation of the Spool Driver 3-18
    - Login and Initialization 3-18
    - Spooling Parameters 3-19
    - To Continue Spooling 3-20
    - To Terminate Spooling 3-21
    - Spool Driver Messages 3-21
    - Spool Driver Commands 3-21
  - Operation of a Remote Driver 3-21
    - Initializing and Dialing In the Remote Station 3-22
    - Sending a QUIT Signal To The Driver . 3-23
- Driver Command Descriptions 3-23
  - auto_start_delay 3-25
  - banner_bars 3-25
  - banner_type 3-26
  - cancel 3-26
  - clean_pool 3-27
  - copy 3-27
  - ctl_term 3-28
  - defer 3-29
  - defer_time 3-29
  - go 3-30
  - halt 3-30
  - help 3-31
  - hold 3-31
  - inactive_limit 3-32
  - kill 3-32
  - logout 3-33
  - master 3-33
  - new_device 3-34
  - next 3-34
  - paper_info 3-35
  - pause_time 3-36

		Page
	print	3-37
	prt_control	3-37
	punch	3-38
	pun_control	3-39
	read_cards	3-40
	ready	3-40
	reinit	3-41
	release	3-41
	req_status	3-42
	restart	3-43
	restart_q	3-44
	runout_spacing	3-44
	sample	3-46
	sample_form	3-47
	sample_hs	3-47
	save	3-48
	sep_cards	3-49
	single	3-50
	slave	3-50
	slave_term	3-51
	start	3-52
	station	3-52
	status	3-53
	step	3-54
	x	3-54
Section 4	Management of Card Input Station	4-1
	Card Input Access Control	4-1
	Card Input Password	4-1
	Registering Card Input Users	4-1
	Remote Job Entry Submission Access Control	4-1
	Station Registration and Password	4-1
	User Card Input Access Segment	4-2
	System Station Access Control Segment	4-2
	Reading User Card Decks	4-2.1
	Reading Cards at the Central Site	4-4
	Login and Initialization	4-5
	Communicating with the Daemon	4-5
	Error Conditions	4-6
	Reading Cards at the Remote Site	4-6
Appendix A	Administrative Commands and Active Functions	A-1
	create_daemon_queues, cdq	A-2
	cv_prt_rqti	A-3
	display_prt_rqti	A-4
	iod_command	A-4.1
	iod_tables_compiler	A-5
	iod_val	A-6
	print_devices	A-7
	print_iod_tables	A-8
	print_line_ids	A-9
	print_spooling_tape	A-10
	Description of the Spooling Tape	A-11
Appendix B	Summary of I/O Daemon Commands	B-1
	Standard Driver Commands	B-1
	Device Specific Driver Commands	B-3
	Commands for Printers	B-3
	Commands for Printers at Request Command Level Only	B-3
	Commands for Local Punches	B-4
	Commands for Remote Punches	B-4
	Command for Remote Punches at Request Command Level Only	B-4

CONTENTS (cont)

		Page
	Commands for Card Input	B-4
	Commands for Control Terminal Operation (Most Drivers)	B-4
	Commands for Remote Device Control . . .	B-4.1
	Commands for Spool Driver	B-4.1
Appendix C	I/O Daemon Admin exec_com Format	C-1
Appendix D	Generating A Driver Process In Test Mode . . .	D-1
	Test Directory Structure	D-1
	User Generated Data Bases	D-1
	Shared Data Bases	D-2
	Manipulating Requests in the Test Queues . .	D-2.1
	The Test Process	D-3
	Testing a Remote Station	D-3
	Setting Breakpoints	D-4
	Command Level Messages	D-5
	Sample exec_com File	D-6
	Test Mode Commands	D-7
	coord	D-7
	debug	D-8
	driver	D-8
	pi	D-9
	resume	D-9
	return	D-10
Appendix E	Driver to Driver Message Facility	E-1
Appendix F	IO Modules For Remote Stations	F-1
	hasp workstation I/O Module	F-1
	tty printer I/O Module	F-1
Appendix G	The Hasp Workstation Simulator	G-1
	Simulator Structure	G-1
	Definition of a HASP Workstation Simulator .	G-2
	Iod_tables	G-2
	Sample iod_tables Definition	G-3
	args Statement Keywords	G-3
	minor_args Statement Keywords	G-4
	Operating a HASP Workstation Simulator . . .	G-5
	Special Instructions For Running The Printer and Punch Simulators	G-5
	receive	G-6
	auto_queue	G-7
	request_type, rqt	G-8
	hasp_host_operators_console, hhoc	G-9
Index	. . . .	i-1

SECTION 1

INTRODUCTION

The bulk input/output facility is normally used to manage all card reading, card punching, and printing requests on both local and remote unit record equipment. Printing facilities include a set of priority queues for requests submitted by users, the management of one or more printers, the handling of special forms, and numerous operator commands to control the operation of this facility. An optional operational mode allows the spooling of print requests onto tape for subsequent printing on either the same or another system. The card input facilities include both the input of data and the input and submission of absentee jobs. This facility is integrated with the Multics access control mechanism and the access isolation mechanism (AIM) so that integrity of users' data is maintained. Accounting is provided for bulk input/output.

The software that handles printing, punching, and card input is called the I/O daemon. It normally runs with highly privileged access on the SysDaemon project, though some of the drivers can run with fewer privileges if the site desires. The I/O daemon is organized into a coordinator process and a number of driver processes; a driver is associated with each local or remote device.

The I/O daemon normally is run with message coordinator terminals. The particular terminal or terminals chosen depend upon the needs of the site. For remote devices, partial control of the process is from the device itself, using its card reader or input keyboard if available.

The system administrator sets up the environment in which this facility runs by creating and modifying the I/O daemon data bases, creating info segments or other information to inform the user community of what is available, and setting up special operator exec_coms or instructions. The operator runs the facilities according to instructions given by the system administrator, taking care of the needs of the peripheral devices and following special requests made by the system.

Section 2 of this manual describes the directory structure and data bases used by the I/O daemon. System administrators must be familiar with the material in this section in order to set up the I/O daemon to meet the needs of their particular site.

Section 3 describes the operational capabilities of the I/O daemon and the commands and operating procedures needed to make use of these capabilities. Both the system administrator and the operators should be familiar with the material in this section in order to provide the best possible service. Under normal operational circumstances, the operator uses the special commands contained in the system exec_coms described in the MOH.

Section 4 describes the data bases that a system administrator uses to manage a card input station in a manner that ensures against security violations. The control cards that the operator places at the beginning and end of each user supplied card deck are described here, as well as the special operating instructions used, in conjunction with those in Section 3, to operate the card reader.

Appendix A describes the administrator commands used to manage the I/O daemon data bases and the operator command used to print a spooling tape.

Appendix B contains a summary of all I/O daemon commands. For convenience, copies of this section should be made and posted near control terminals used by the I/O daemon.

Appendix C explains I/O daemon admin exec_coms, with special reference to the driver x command. A sample section of an admin exec_com completes this appendix.

Appendix D details how to generate an I/O daemon driver process in a test environment.

Appendix E describes how to set up a message facility for communication between drivers and from drivers to devices.

Appendix F reproduces, from the first edition of this manual, the original method of filling out the I/O daemon tables for remote driver devices.

Appendix G describes a facility for the simulation of a remote job entry (RJE) workstation using the HASP communications protocol.

SECTION 2

DIRECTORY STRUCTURE AND DATA BASES OF THE I/O DAEMON

I/O DAEMON DIRECTORIES

The Multics I/O daemon software depends on the existence of certain directories and segments. The most important of these directories and segments are created and initialized at a new Multics site by the acct_start_up.ec segment (described in MAM System).

The system administrator who manages and supervises the various I/O daemon processes must be familiar with that portion of the hierarchy around which the daemon processes are organized. The main node of this hierarchy is the directory named >daemon_dir_dir (with a short name of >ddd). This directory contains segments and directories used to support the various system daemon processes.

Contents of daemon_dir_dir Directory

The >daemon_dir_dir directory contains the following directories of interest:

io_daemon_dir	holds all I/O daemon data bases
cards	a storage pool for card deck image segments read by the system card input process (local or remote station)
io_msg_dir	contains mailboxes for each device (station) for which driver to driver messages will be sent or received

These directories and their contents are described in the following paragraphs. The access isolation mechanism (AIM) access class for all these directories is system low.

Contents of io_daemon_dir Directory

The >daemon_dir_dir>io_daemon_dir directory contains a set of administrative data bases and working storage used to direct the activities of the I/O coordinator process and the various device drivers.

The main data base, iod_tables, is set up by a system administrator. Most of the other segments and directories are created and maintained by the I/O coordinator acting on information contained in the iod_tables segment.

The following segments are contained in io_daemon_dir:

coord_comm.ms — ring 1 message segment in which driver processes place messages for the I/O coordinator

coord_lock — a segment used by the process overseer of IO.SysDaemon to prevent initialization of a driver process before an I/O coordinator has been created; also prevents creation of more than one I/O coordinator

iod_tables — master control tables for the I/O coordinator; compiled by the iod_tables_compiler command on request of a system administrator

iod_tables.iodt — source segment for iod_tables; may be updated by a system administrator and compiled to yield iod_tables

iod_working_tables — working copy of iod_tables used by the device drivers and users; copied from iod_tables during I/O coordinator initialization

iodc_data — a segment containing the process identifier of the I/O coordinator and the event channel identifier to be used by a new driver for initial communication with the coordinator

printer_notice — an optional segment containing information that the site administrator wants to be printed on the page following the head sheet of every printer listing, local or remote. The segment must contain ASCII text. It should be no longer than 60 lines and the line length should correspond to the shortest printer device in use. This feature is useful in notifying users of printing charge rates, available request types, when they are processed, stock forms used for each, and other useful information covering the printing operations on the system.

XXX_N.ms — ring 1 message segment for I/O daemon queues; one such message segment is created for each priority queue of a request type; XXX is the request type name and N is the queue number (1<= N <=4)

There are several directories contained in the io_daemon_dir, some of which are site dependent. Access class information about these directories is included here as an aid to those sites using the access isolation mechanism (AIM).

coord_dir — working storage for the I/O coordinator, which is created and managed by the I/O coordinator; the access class is the authorization of the I/O coordinator

rqt_info_segs — created by an administrator to hold any request type info segments used by drivers (see the cv_prt_rqti command in Appendix A and "Request Type Info Segments" later in this section); the access class is system low

meter_dir — created by an administrator to hold driver metering data (for future use)

<major device> — separate directory for each major device currently being run by a device driver. These directories are managed by the I/O coordinator and their names are site dependent. Each major device directory contains a driver status segment for each minor device associated with the major device. The access class is the authorization of the device driver.

This page intentionally left blank.

Contents of cards Directory

The storage pool for card deck image segments consists of a subtree of the directory hierarchy, which is headed by >daemon_dir_dir>cards. One access class directory for each access class (as needed) is contained in the cards directory. Storage is always allocated within the access class directory that corresponds to the process' authorization. Person directories are contained in the appropriate access class directory. A person directory is created for each person who needs temporary storage. A person directory contains all segments and multisegment files for a given person at a given access class. For example, if a user with Person_id TSmith is at system_low, the following directory is allocated for his card deck image segments:

>daemon_dir_dir>cards>system_low>TSmith

Contents of io_msg_dir Directory

The >daemon_dir_dir>io_msg_dir directory contains mailboxes of the form <device>.mbx for each device and remote station that uses the driver to driver message facility. See Appendix E for more information.

I/O DAEMON TABLES

In order to manage the use and operation of the I/O daemon, an administrative data base exists that can be adapted to the specific needs of a particular Multics site. This data base contains several different tables of information and hence is referred to as the "I/O daemon tables." The data base is generated from a source language description ordinarily prepared by a system administrator. The iod_tables_compiler command (described in Appendix A) is used to translate the source description into the encoded representation of the I/O daemon tables. The encoded representation, which is used by the I/O coordinator, must be named "iod_tables".

I/O Daemon Tables Source Language

The purpose of the I/O daemon tables source language is to define the devices and the request types to be used by the I/O daemon. A source file consists of a sequence of statements and substatements that define and describe each device and request type. In addition, certain global information items are defined that do not pertain to any particular device or request type.

SYNTAX

The syntax of the source language statements and substatements is of the form:

<keyword>: <parameter>;

The only exception to this is the "End" statement. The keyword of a statement begins with a capital letter; the keyword of a substatement is entirely in lowercase letters. Substatements describe attributes of devices, communication lines, or request types for the given statement. Each group of one statement and its substatements constitutes a statement description.

PL/I style comments beginning with "/*" and ending with "*/" may appear anywhere within the source file. Similarly, blanks, tabs, and newlines not embedded within a keyword or parameter are ignored. However, in order to include blanks, tabs, newlines, colons, or semicolons in a parameter enclose them in quotes. If a parameter begins with a quote, all immediately following characters up until the next quote are taken as the parameter. It is possible to embed quotes within a quoted string using the double quoting escape convention of PL/I.

STATEMENTS

The following statements may appear anywhere within the source file.

Time: <number>;
: defines the number of minutes that the coordinator saves a processed request. When segment deletion is requested, it is delayed this amount of time. If some problem is discovered that necessitates the reprocessing of requests, those requests performed less than <number> minutes ago can be restarted. One, and only one, Time statement must appear in the file.

Max_queues: <number>;
: defines the default number of priority queues for each request type. The maximum value of <number> is 4. (Queue 1 is the highest priority and queue 4 is the lowest priority.) One, and only one, Max_queues statement must appear in the file.

Line: <name>;
: defines the name of a logical line_id and denotes the beginning of a line description. Any subsequent substatements (see below) apply to this line until the next Line, Device, or Request_type statement is encountered. Any <name> may be chosen; it can be a maximum of 32 characters and cannot contain spaces or periods. There may be up to 360 Line statements. This statement is optional.

Device: <name>;
: defines the name of a major device and denotes the beginning of a device description. Any subsequent substatements (see below) apply to this device until the next Line, Device, or Request_type statement is encountered. Any <name> may be chosen; it can be a maximum of 24 characters and cannot contain periods or spaces. At least one Device statement must appear in the file.

Request_type: <name>;
: defines the name of a request type and denotes the beginning of a request type description. Any subsequent substatements (see below) apply to this request type until the next Line, Request_type, or Device statement is encountered. Any <name> (not containing periods or spaces) may be chosen; it can be a maximum of 24 characters. Currently, however, the names of request types used by the dprint or dpunch commands are restricted to a maximum of eight characters. At least one Request_type statement must appear in the file.

End;
: marks the end of the source language description. Unlike all other statements, it has no parameter. Any text occurring beyond the End statement is ignored. One, and only one, End statement must appear in the file.

SUBSTATEMENTS FOR LINES

The following substatements describe various attributes of a line and may appear in any order following a Line statement.

channel: <name>;
: defines the name of the communications channel to be attached when using the logical line_id (defined in the Line statement). It is normally a teletype channel identifier for an RJE station. The <name> may be up to 32 characters and cannot contain any spaces. One, and only one, channel substatement must be given for each Line statement.

att_desc: <string>;
: defines the attach description to be passed to the remote_teleprinter_ I/O module. The <string> may be up to 256 characters and should appear in quotes since there will be imbedded spaces. If the control variable ^a appears in <string> it will be replaced by the channel <name> (described above). One, and only one, att_desc substatement must be given for each Line statement.

device: <name>;
: defines a major device that can use this logical line_id. At least one device substatement must be given for each Line statement. Any major device specified must also have the line: variable; substatement under the Device statement.

SUBSTATEMENTS FOR DEVICES

The following substatements describe various attributes of a device and may appear in any order following a Device statement.

driver_module: <name>;
: defines the name of a procedure to be executed by a driver process when running the associated device. The <name> can be a full pathname or simply an entryname. In the latter case, the search rules are used to locate the procedure. Several standard driver modules are provided by the system (see "Standard Driver Modules" below). One, and only one, driver_module substatement must be given for each Device statement.

default_type: <name>;
: defines the default request type for the associated device. The <name> must appear as the parameter in a Request_type statement. Unless overridden by the operator when a driver is initialized, the driver processes requests of this default type.

args: <string>;
: defines an argument string to be interpreted by the driver module for the associated device. The <string> may have any arbitrary format up to a maximum of 256 characters. In practice, the composition of the <string> depends on the particular driver module that interprets it. Each driver module has its own conventions for the <string> format (see "Standard Driver Modules" below).

The following three substatements describe alternate methods by which a driver may attach the associated device. These substatements are mutually exclusive. One, and only one, of these substatements must be given for each device statement.

prph: <name>;
names an input/output multiplexer (IOM) peripheral channel through which the associated device can be attached. The <name> must appear on a PRPH card in the configuration deck.

line: <name>;
names a dedicated communications line channel through which the associated device can be attached. If <name> is "variable" the channel can be any logical line_id defined by a Line statement. The driver process must have the dialok attribute in the process definition table (PDT) and the communications channel must be defined as slave in the channel definition table (CDT). See MAM Communications, Order No. CC75, for more information about the PDT and the CDT.

dial_id: <name>;
defines the dial identifier to be used if the associated device is to be dialed to the driver process over a communications line.

The following three substatements describe alternate methods by which the driver may attach a control terminal. These statements are mutually exclusive. If none is specified, the driver assumes that no control terminal is desired.

ctl_line: <name>;
names a dedicated communications line channel through which the control terminal can be attached. The driver process must have the dialok attribute in the process definition table (PDT) and the communications channel must be defined as slave in the channel definition table (CDT). See MAM Communications, Order No. CC75, for more information about the PDT and the CDT.

ctl_dial_id: <name>;
defines the dial identifier to be used if the control terminal is to be dialed to the driver process over a communications line.

ctl_source: <name>;
defines a message coordinator source name to be associated with the driver. (The message coordinator is described in the MOH). A single control terminal accepted by the message coordinator can be used to control many different drivers.

SUBSTATEMENTS FOR REQUEST TYPES

The following substatements describe various attributes of a request type and may appear in any order following a Request_type statement.

accounting: <name>;
defines the name of an accounting procedure to be executed by a driver when processing requests of the associated type. The <name> can be a full pathname or simply an entryname. In the latter case, the search rules are used to locate the procedure. Also, the special <name> "system" can be used to indicate the standard system accounting procedure. If this substatement is omitted, "system" accounting is assumed.

card_charge: "p1, p2, p3, p4";

defines the resource price names for the card_charge of each queue of the request type. This substatement is optional. If it is not specified, the prices from system_info_$io_prices are used. p1 through p4 are resource price names, which are defined using the ed_installation parms command. The prices must be defined before the iod_tables segment is compiled, or the compilation will fail. The price names for each queue must be given in order, from queue 1 to the maximum number of queues for the Request type description. Each price is defined in units of dollars per 1000 cards. Note: card_charge must not be specified if line_charge is specified.

default_queue: <number>;

The default_queue substatement is used to define the default queue for a request type. The value of <number> may be from 1 to max_queues. If not specified, it is set to the value defined in the max_queues substatement, but it will not be greater than 3.

device: <name>;

specifies a device that can be used to process requests of the associated type. The <name> must appear as a parameter in a Device statement. More than one device substatement may be specified for a request type.

driver_userid: <access_name>;

defines the required person and project names for a driver of the associated request type. If omitted, the <access_name> defaults to IO.SysDaemon, which is the standard system driver. Other access names may be used, for example, to provide a project with its own private driver.

This page intentionally left blank.

generic_type: <name>;
: defines the generic type of the associated request type. If the generic type name matches the request type name, then the request type is the default for the generic type. One, and only one, generic_type substatement must be given for each Request_type statement. If the generic type is neither "printer" nor "punch", the list_daemon_requests command (see the MPM Commands) must be used with the -brief control argument. The cancel_daemon_requests command cannot be used.

line_charge: "p1, p2, p3, p4";
: The line_charge substatement defines the resource price names for the line charge of each queue of the request type. This substatement is optional. If not specified, the prices from system_info_$io_prices will be used. If specified, each price name must be defined in the system price table, or the compilation of the the iod_tables will fail. The price names for each queue must be given in order, from queue 1 to the maximum number of queues for the Request_type description. Each price is defined in units of dollars per 1000 lines. *

max_queues: <number>;
: The max_queues substatement may be used to define the maximum number of queues for a request type, when it is different from the global Max_queues value. This substatement is optional. The value of <number> may be from 1 to 4.

page_charge: "p1, p2, p3, p4";
: The page_charge substatement defines the resource price names for the page charge of each queue of the request type. This substatement is optional. If it is not specified, the prices from system_info_$io_prices are used. If specified, each price name must be defined in the system price table, or the compilation of the iod_tables fails. The price names for each queue must be given in order, from queue 1 to the maximum number of queues for the Request type description. Each price is defined in units of dollars per 1000 pages.
Note: page_charge must not be specified if generic type is "punch".

rqti_seg: <name>;
: The rqti_seg substatement is used to define the name of the request type info (rqti) segment to be used with the Request_type statement. This substatement is optional. When specified, <name> must correspond to a segment entryname in the >ddd>idd>rqt_info_segs directory, or the driver will fail initialization. When not specified, no driver will look for an rqti segment for this Request_type statement.

SOURCE FILE EXAMPLE

The subset of the source language described so far is sufficient to prepare a complete I/O daemon tables source file. Many sites will find they have no need of any other features. Thus, before describing some of the less commonly used statements and substatements, an example of a source file containing just the ones described above is presented.

```
/* Example of an I/O daemon tables source file */

/* Global parameters */

Time:       60;   /* save requests for 60 minutes */
Max_queues: 3;    /* 3 priority queues per request type */
```

```
/* Devices */

Device:              printer_1;      /* onsite printer */
  driver_module:     printer_driver_;
  prph:              prta;
  default_type:      printer;

Device:              punch_1;        /* onsite punch */
  driver_module:     punch_driver_;
  prph:              puna;
  default_type:      punch;

/* Request types */

Request_type:        printer;   /* onsite printer requests */
  generic_type:      printer;
  device:            printer_1;

Request_type:        punch;     /* all punch requests */
  generic_type:      punch;
  max_queues:        1;
  default_queue:     1;
  line_charge        punch_price;
  device:            punch_1;

End;
```

In the sample source file there are two devices and two request types. The request types are handled by the standard system driver, IO.SysDaemon, as implied by the absence of any driver_userid substatement. These two request types, printer and punch, are the default types for the dprint and dpunch commands respectively. The majority of users concern themselves only with these two request types. Output is produced onsite by the printer_1 and punch_1 devices.

MAJOR AND MINOR DEVICES

Special provisions have been made to handle "combination" devices that contain more than one logical device (e.g., a printer and a punch) in a single physical unit. The combination device as a whole is referred to as a "major device"; the multiple subdevices, such as a printer and a punch, are referred to as "minor devices". A major device is connected to Multics via a single communications channel; it can be attached by one process only. Therefore, it is not possible to have separate driver processes running the separate logical devices. To overcome this problem, the driver software has been designed to simulate multiple drivers within a single process. This means that from the coordinator's point of view, each logical device is distinct and run by an independent driver process. Consequently, each one of these logical devices can be fed requests of a different request type and generic type.

Major devices are defined by the Device statement described earlier. Similarly, minor devices are defined by a minor_device substatement. The minor_device substatement is treated as a substatement for devices and, as such, can be freely intermixed with other substatements for devices.

minor_device: <name>;
: defines the name of a minor device belonging to the associated major device and denotes the beginning of a minor device description. Any subsequent substatements (see below) apply to this minor device until the next minor_device substatement or Line, Device, or Request_type statement is encountered. Any <name> may be chosen up to a maximum of 24 characters.

If no minor devices are explicitly defined for a major device, then a default minor device is defined by implication. The primary purpose of a default minor device is to allow certain minor device substatements to be specified for a major device when it has no explicit minor devices. One such substatement is the default_type substatement that was previously described under "Substatements for Devices." In fact, the default_type substatement is actually a substatement for a minor device. When no minor devices are explicitly defined, the default_type substatement applies to the default minor device of the preceding major device. The same is true of all substatements for minor devices.

SUBSTATEMENTS FOR MINOR DEVICES

The substatements below describe attributes of a minor device and may appear in any order following a minor_device substatement or following a Device statement if no minor devices are specified.

minor_args: <string>;
: defines an argument string to be interpreted by the driver module for the associated major device. The string may have any arbitrary format up to a maximum of 256 characters. Conventions for the <string> format expected by standard system driver modules are described under "Standard Driver Modules" later in this section.

default_type: <name>;
: defines the default request type for the associated minor device. The <name> must appear as a parameter in a Request_type statement.

The device substatement described earlier is a substatement for a request type and is used to name devices that can process requests of a given type. Usually, the parameter of a device substatement is a major device name. However, if minor devices are defined for the major device, then the device substatement parameter must include both the major and minor device names separated by a period (e.g., xyz.printer).

SOURCE FILE EXAMPLE USING MINOR DEVICES

This example shows a portion of a source file that illustrates the use of minor devices.

```
Device:             xyz; /* a combination device */
  driver_module:    dummy_driver_;
  args:             "dim= xyz";
  line:             a.h100;
minor_device:       printer;
  minor_args:       "dev= printer";
  default_type:     xyz_prt;
minor_device:       punch;
  minor_args:       "dev= punch";
  default_type:     xyz_pun;

Request_type:       xyz_prt;
  generic_type:     printer;
  device:           xyz.printer;

Request_type:       xyz_pun;
  generic_type:     punch;
  device:           xyz.punch;
```

AIM FEATURES

The I/O daemon incorporates certain features in support of the access isolation mechanism (AIM). System administrators at sites not using authorizations above system_low need not read the following and should instead skip "Standard Driver Modules" below.

Every request processed by the I/O daemon has an access class. The access class of a request is equal to the authorization of the process that submitted the request. Each piece of output normally has an access class banner. For print requests, the access class banner appears on the head sheet of each printout. For punch requests on the local punch, the access class banner appears in the flip cards at the beginning of each deck. At remote sites, no access class banner appears. However, if the access class of a request is system_low and the access class name for system_low is null, then the access class banner is omitted.

In the interest of security, some sites may find it desirable to have requests of the same type automatically separated according to access class. To illustrate how this access class separation might be used, imagine a site at which two different access classes are defined. One of these, called "public," is available to all users. The other, called "confidential," is available to only a limited number of users who deal with sensitive information. Further, suppose that the site has two printers, both of which are used to process requests of the same type. Assume that different distribution points for public and confidential output exist so that stricter control can be exercised over the release of confidential output. In this case, the operators must separate confidential output from public output by examining the access class banners. An error in bursting or separating the output could result in confidential output being accidentally released with public output. In addition to this security weakness, there is also the operational burden of separating the output according to access class.

The I/O daemon offers a solution to the above problems. Each driver process can be made to handle only requests of a single access class or a range of access classes. Therefore, all public output could be directed to one printer and all confidential output to the other. Hence, both the operational burden and the potential for operational errors mentioned above are eliminated. To facilitate output handling, the public printer could actually be located in the public distribution area while the confidential printer could be located in the confidential distribution area.

The benefits of this automatic separation are not obtained without cost. It is probable that printer utilization and hence turnaround time for output will be somewhat degraded on the whole. This is because it is unlikely that the amount of output will be evenly divided between the access classes. For example, the number of public requests might be much larger than the number of confidential requests. In this case, the confidential printer would be underutilized.

Other disadvantages become evident if one considers the situation where there are fewer printers than access classes. If instead of two printers, only one were available at the hypothetical site, then this printer would have to be switched back and forth between public and confidential output. This switching, of course, increases the operational burden. Also, it upsets the priority selection of requests. Suppose, for example, that the site decides to switch between public and confidential output every 30 minutes. A print request submitted to queue 1 might have to wait this amount of time before being processed. By contrast, if the printer were processing both access classes at once, the request would be performed immediately (assuming queue 1 were empty).

Unfortunately, even with the switching of printers from one access class to another, automatic access class separation of output simply does not scale up for a large number of access classes. Clearly, at some point it becomes impractical to rotate a small number of devices among a larger number of access classes. Therefore, sites using a large number of access classes or sites not willing to tolerate some of the drawbacks cited above may choose to forego automatic access class separation of output. In this case, each device can be made to handle the full range of access classes from system_low to system_high. Care must be taken to ensure proper distribution of output. Control forms can provide a helpful receipt for each piece of output.

The mechanism for separating output according to access class is the "device class." Each request type can be partitioned into any number of separate device classes. One or more devices can be specified for each device class. Also, a range of access classes can be specified for each device class. When a driver process is initialized, the operator normally indicates the device to be run and the request type. However, if device classes are defined for the request type, then the operator must also indicate a device class. This determines the access class range of requests that the driver processes.

It is important to note that the device class of a request is not something the user can specify. In fact, the entire device class concept is invisible to users. Unlike the type and priority queue of a request, the device class is not determined at request submission time. Rather, it is determined at request processing time. Hence, it is possible to modify the I/O daemon tables and change the predicted device classes of requests stored in the queues.

A device class is defined by a device_class substatement. The device_class substatement is treated as a substatement for request types and, as such, can be freely intermixed with other substatements for request types.

device_class: <name>;

defines the name of a device class belonging to the associated request type and denotes the beginning of a device class description. Any subsequent substatements (see below) apply to this device class until the next device_class substatement or Request_type, Line, or Device statement is encountered. Any <name> may be chosen up to maximum of 24 characters.

If no device classes are explicitly defined for a request type, then a default device class is defined by implication. The primary purpose of a default device class is to allow certain substatements for device class to be specified for a request type when it has no explicit device classes. One such substatement is the device substatement that was previously described under "Substatements for Request Types." In fact, the device substatement is actually a substatement for a device class. When no device classes are defined, the device substatement applies to the default device class for the preceding request type. The same is true of all substatements for device classes.

SUBSTATEMENTS FOR DEVICE CLASSES

The substatements below describe various attributes of a device class and may appear in any order following a device_class substatement or following a Request_type statement if no device classes are defined.

min_access_class: <access_class>;

defines the minimum access class of a request to be processed in the associated device class. The <access_class> must be a standard access class string as defined by the convert_authorization_ subroutine. If omitted, the default minimum is system_low.

max_access_class: <access_class>;
defines the maximum access class of a request to be processed in the associated device class. The <access_class> must be a standard access class string. If omitted, the default maximum is the access_class string given in min_access_class.

min_banner: <access_class>;
defines the minimum access class banner to be placed on the head sheet of printed output, on the flip cards of punched output, and on the control forms for all output. Normally, the access class of the request is used. However, if this access class is less than that specified for min_banner, then the min_banner value is used. The <access_class> must be a standard access_class string. If omitted, the default min_banner is the access_class string given in min_access_class.

device: <name>;
specifies a device that can be used to process requests of the associated device class. The <name> must appear as the parameter of a Device statement. More than one device substatement may be specified for a device class.

Care should be taken to ensure that the full system access range (system_low to system_high) is covered by the union of access ranges of the device classes for each request type. (If no device classes are defined for a request type, the max_access_class substatement should be set to system_high for the default device class.) If not, requests of access classes that are not included are never processed. Upon discovering such a request, the I/O coordinator prints an error message and skips the request. Also, it should be noted that if two or more device classes from the same request type have overlapping access ranges, then a request falling in this overlap is assigned to the device class defined first in the I/O daemon tables source file.

As mentioned above, when multiple device classes are defined for a request type, requests are generally not performed in the usual order dictated by priority and submission time. This phenomenon is most noticeable when one device must be shared among several device classes. In order to aid the operators in determining when to switch a device to a different device class, the I/O coordinator keeps track of "waiting" requests. A waiting request is one that is passed over in the normal request selection order while the coordinator looks for a request to satisfy a different device class, or is explicitly requested to run at high priority by an operator command. A count of waiting requests is kept on a per device class basis. When the number of waiting requests for a device class becomes large, this indicates that the device class is receiving inferior service relative to some other device class for the same request type. Thus, operators could be instructed to switch a device to another device class whenever the number of waiting requests reaches some limit. (See the coordinator command, wait_status, in Section 3.)

SUBSTATEMENT FOR DEFAULT REQUEST TYPE

The default_type substatement described earlier under "Substatements for Devices" names the default request type that a device processes unless overridden by the operator. However, if device classes are defined for the request type, then the parameter of the default_type substatement must include both the request type and device class names separated by a period (e.g., printer.confidential).

SOURCE FILE EXAMPLE USING AIM

This example shows a portion of a source file that illustrates the use of AIM features.

```
Request_type:         printer;
  generic_type:       printer;

device_class:         public;         /* for system_low output */
  device:             printer_1;      /* primary public printer */
  device:             printer_2;      /* can use this one in
                                         emergencies*/

device_class:         confidential; /* for output above system_low */
  min_access_class:   level1;
  max_access_class:   system_high;
  min_banner:         level2;         /* all confidential output
                                         has at least a level 2 banner
                                         authorization */

  device:             printer_2;       /*  use only this printer located
                                         in secure area */

Device:               printer_1;
  driver_module:      printer_driver_;
  prph:               prta;
  default_type:       printer.public;

Device:               printer_2;
  driver_module:      printer_driver_;
  prph:               prtb;
  default_type:       printer.confidential;

Request_type:         punch;
  generic_type:       punch;
  max_access_class:   system_high;     /* handle all access classes */

Device:               punch_1;
  driver_module:      punch_driver_;
  prph:               puna;
  default_type:       punch;
```

Standard Driver Modules

A driver module must be specified for each device defined in the I/O daemon tables. A driver module is a program that embodies specific knowledge of how to manipulate a particular device. The standard driver modules provided by the system are described below.

As mentioned earlier in this section, the <string> argument of the args or minor_args substatements are interpreted by each individual driver module. Even though the format of these strings is defined as arbitrary, each of the standard driver modules support a basic <string> having the following syntax:

```
key= value
```

The key must be unique in <string> and acts like a control argument. The value is the argument associated with the key. Keys and values may not contain commas, but may contain spaces. The key/value pairs are separated from one another by a comma. For example:

```
args:      "dim= device_dim_, form_type= xxx";
```

The complete <string> must appear in quotes and standard Multics quoting conventions apply within <string>. The total length of <string> cannot exceed 256 characters.

The following paragraphs describe the args and minor_args keys that are supported by each of the standard driver modules, as well as other attributes of the I/O daemon tables device specification.

printer_driver_ MODULE

This driver module should be specified for standard Multics printers. The prph substatement must be specified for the associated device. Multiple minor devices are not supported and the minor_args substatement is ignored. For standard printer operation, no args substatement need be specified. However, the args substatement can be used to define a nonstandard device interface module (DIM) and/or a nonstandard control terminal accountability form type. This is done by including the following key-value pairs in the args substatement.

dim= <DIM_name>

The "dim=" key defines <DIM_name> to be the DIM through which the device is attached. The default DIM for printer_driver_ is prtdim_.

form_type= <form_name>

The "form_type=" key defines <form_name> to be the control form type. If not specified, a default control form type is used.

punch_driver_ MODULE

This driver module should be specified for standard Multics punches. The prph substatement must be specified for the associated device. Multiple minor devices are not supported and the minor_args substatement is ignored. For standard punch operation, no args substatement need be specified. However, punch_driver_ accepts an args statement of the same form as printer_driver_. The default DIM is cpz.

reader_driver_ MODULE

This driver should be specified for standard Multics card readers. The prph substatement must be specified for the associated device. Multiple minor devices are not supported and the minor_args substatement is ignored. For standard reader operation, no args substatement need be specified. However, the following key-value pairs may be specified in the args substatement:

dim= <DIM_name>

The "dim=" key defines <DIM_name> to be the DIM through which the device is attached. The default dim for reader_driver_ is crz.

station= <Station_id>

The "station=" key defines <Station_id> to be the name of the card input station to be associated with this card reader. The default station id is "reader". For example:

```
Device:           reader;
 driver_module:   reader_driver_;
 prph:            rdra;
 default_type:    dummy;

Request_type:     dummy;
 generic_type:    dummy;
 max_queues:      1;
 device:          reader;
```

While the reader_driver_ does not process requests from the coordinator, the syntax of the iod_tables requires the presence of a request_type substatement. This should be a dummy request type to which no users have access to submit requests, as for the reader minor device of the remote_driver_.

Sites with CCU (combined card unit) devices should define two devices: one with punch_driver_ for the punch, and one with reader_driver_ for the reader.

spool_driver_ MODULE

This driver module should be specified for a major device that will be used to write user print requests onto tape instead of the printer. The prph substatement must be specified, but the <name> need not be an IOM channel. (It is used as an I/O switch name for the tape attachment.) The default type may be omitted if the operator is required to specify the request type each time the "device" is used. For example:

```
Device:           spooler;
 driver_module:   spool_driver_;
 prph:            tape;
```

The spool_driver_ ignores all args substatements. It does not accept multiple minor devices and does not accept any control terminal specifications.

This page intentionally left blank.

This driver module should be specified for all remote printer/punch/reader stations. Two types of stations are supported by the remote driver. A Type I station can be initialized from any one of several communications lines. A Type II station, which does not have an input device, is initialized on a dedicated communications line as a predefined station. The two station types are described separately below because the iod_tables description of each is different.

The driver process must have the dialok attribute in the PDT, and it must have the rw access to the access control segment (ACS) of the communications line it will attach. The remote_driver_ can handle one minor device for a card reader and an arbitrary number of minor devices for printers and punches within the limits of the physical remote device and the line protocol. (A minor device for the reader must be specified if the remote device is to read card input.)

The remote_driver_ is designed for maximum flexibility, so its description is rather complex. The reader should examine the entire section before attempting to set up a remote driver in the iod_tables.

Normal setup of the remote_driver_ (Type I stations)

To set up the remote_driver_ for a remote station, the administrator defines a set of communication lines for remote stations, with driver processes listening to each line as stations dial in.

The operator types in the station_id and password via the station command. Once validated, the driver locates the major device that has the same name as the station_id, and begins initialization. If default request types are defined for the minor devices, they are used. If the default request type is omitted for one or more minor devices, the remote station operator is asked to specify the request type. Of course, the minor device must be allowed to use the request type by a device substatement in the Request_type description.

There must be at least one Line statement for each communications line. The Line statement defines the logical line_id and specifies the channel, the attach description (which defines the terminal type), and which stations may use the line_id. For example:

```
Line:                2780_1;
 channel:            a.h001;
 att_desc:           "-tty ^a -terminal ibm2780_ -comm bisync_
                     -ebcdic -ttp IBM2780 -runsp 5 -ttd_limit 2
                     -bretb -multi_record";
 device:             station_a;
 device:             station_b;

Line:                2780_2;
 channel:            a.h002;
 att_desc:           "-tty ^a -terminal ibm2780_ -comm bisync_
                     -ebcdic -ttp IBM2780 -runsp 5 -ttd_limit 2
                     -bretb -multi_record";
 device:             station_a;
 device:             station_b;
```

Each logical line_id (e.g., 2780_1) describes a communications line that a station may dial into. The attach description defines the type of station using the channel. If a single channel (communications line) is to be used for more than one device type, separate line_ids can be defined with the same channel to

allow the central site operator to choose the device type during driver initialization.

The attach description string is the one used to attach the teleprinter device, or console, of the station. It is also the basis for the attachment description of the other minor devices. If the attach description for a minor device is to be different from the teleprinter device for that minor device, the attach options may be put into the minor args following a desc= key. Any attach options found in the minor args will override those of the teleprinter device for that minor device.

Each station_id that may use a given line_id is listed as a device in the device substatement of the Line statement, and each must correspond to a major device in a Device statement.

There must be a Device description specified for each station_id. The Device description must include a line substatement with the keyword "variable" specified. This will allow the driver to use some or all of the communication lines defined in Line statements. There must also be minor_device substatements defined for each device attached to the remote terminal. The default_type substatements may be omitted if the remote station operator is to specify the request type for the minor devices. Normally, the Device description will be general enough to allow the station to run any device type, as shown in the following example. (This might not be true if special attach options are defined for one or more minor devices using the desc= key in the minor_args substatement.)

```
Device:               station_a;
 line:                variable;
 driver_module:       remote_driver_;

 minor_device:        prt;
  minor_args:         "dev= printer";
                      /* no default type has been specified */

 minor_device:        pun;
  minor_args:         "dev= punch";
  default_type:       sta_pun; /* makes this rqt required */
                      /* for this minor_device */

 minor_device:        rdr; /* so we can read cards */
  minor_args:         "dev= reader";
  default_type:       sta_pun;  /* just a dummy entry */

Device:               station_b;
 line:                variable;
 driver_module:       remote_driver_;

 minor_device:        prt;
  minor_args:         "dev= printer";
  default_type:       stb_prt;  /* always true for station_b */

 minor_device:        pun;
  minor_args:         "dev= punch";
  default_type:       stb_pun; /* makes this rqt required */
                      /* for this minor_device */

 minor_device:        rdr; /* so we can read cards */
  minor_args:         "dev= reader";
  default_type:       sta_pun;  /* just a dummy entry */
```

Each station may dial in on either Line (see example above); its operating characteristics will be the same. The Device descriptions for the two stations shown have the following difference. Station_a is allowed to specify its printer request type after giving its station command, but station_b will always

use the stb_prt request type because this request type is specified in a default_type substatement.

The request type that is used for the default type of the reader minor devices is needed to suppress questions to the operator and to satisfy the syntax rules of the iod_tables_compiler. The request type specified can be any existing request type or it can be a dummy request type used for the reader. No requests will ever be sent by the coordinator for the reader.

A Type I station is always assumed to have an input device, which acts as a slave terminal for the driver. As such, any control terminal definitions associated with the major device will be accepted for the preparation of accountability forms only (see "Terminals that Control the Driver"). However, a Type II station may accept a control terminal as a slave terminal if specified.

Setup for stations that cannot input commands (Type II stations)

Because it has no input device, the Type II station can be identified only by the line it dials into. Therefore, the line substatement for the major device specifies the exact channel name to be used. There are no Line statements associated with this type of station.

```
Device:            station_c;
 line:             a.h003;
 driver_module:    remote_driver_;
 args:             "station= station_c, slave= no,
                   desc= -terminal tty_printer_ -comm tty_;

 minor_device:     prt1;
   default_type:   stc_text;
   minor_args:     "dev= printer, desc= -pll 85 -ppl 66
                    -htab -ttp LA120_10061_8X11";

 minor_device:     prt2;
  default_type:    stc_prt;
  minor_args:      "dev=  printer, desc= -pll 140
                    -ppl 88 -htab -ttp LA120_160L_8X11";
```

A default request type should be specified for each minor device. This is done to avoid making the central site operator answer questions from the driver for each minor device during driver initialization and reinitialization.

Remote Driver <string> Arguments

All the <string> arguments acceptable to the args and minor_args substatements that are defined for the remote_driver_ are described as follows:

a. Arguments which apply to both Type I and Type II stations.

This page intentionally left blank.

desc= <attach_description>

The desc= key is used to specify additional parameters to the I/O module in the form of an iox_ attach description. This key may be used for any minor_args substatement. Any attach options specified will override attach options for the teleprinter device and/or any attach options which are common to all minor devices. This key is also used in the args substatement for Type II stations, to specify attach options for the teleprinter device. For more information, see the definition of the attach description for the communications module or terminal module associated with the major or minor devices. Also, see the attach options for these device modules: remote_teleprinter_, remote_printer_, remote_punch_, and remote_reader_. For descriptions of these modules and other I/O modules, refer to the MPM Communications Input/Output Order No. CC92.

dev= <minor_device_type>
The dev= key is used to specify the device type of a particular minor device. This key is required for each minor device. The value of minor_device_type must be printer, punch, or reader.

form_type= <ctl_term_form_type>
The form_type= key is used to specify the name of a control terminal accountability form. This is an optional argument and is used for major devices only.

b. Arguments which apply only to Type II stations.

*

station= <station_id>
When the station= key is used in an args substatement, the driver will accept any station_id (other than blank). The driver will accept any device dialing in on this channel as this station_id without authentication controls. All specified minor devices and default request types will be used. Normally, the value will be the name of the Device (i.e., the station name). This is used for a station without an input device or with a dedicated communications line.

slave= <yes_or_no>
The slave= key value of "yes" is used to tell the driver that it should accept commands from the remote terminal as a slave terminal, as well as from the central site terminal (master terminal). The slave= yes argument can be used to make a Type II station into a Type I station over a dedicated phone line. This key is optional and is only used in the args substatement of the major device. The default is "no".

*

CREATION AND MAINTENANCE OF I/O DAEMON TABLES

Creation of the I/O daemon tables begins with the preparation of a source segment using the language described earlier in this section. This source segment can be produced with any text editor. As mentioned earlier, the source segment is translated into a binary representation by the iod_tables_compiler command (see Appendix A). By convention, this command assumes that all source segments have a name ending with the iodt suffix. The standard name for the I/O daemon tables source segment is iod_tables.iodt. When this segment is compiled, an object segment is created with the name iod_tables.

The I/O coordinator looks for the iod_tables segment during its initialization. It expects to find this segment in the directory >daemon_dir_dir>io_daemon_dir. (Normally, the source segment is kept in this same directory, although it is not essential.) The I/O coordinator also looks for a second segment named iod_working_tables. This segment is the working copy of the I/O daemon tables and is the one referenced by driver processes and user processes. The reason for this second segment is to facilitate making changes to the I/O daemon tables. Clearly, the source segment can be modified at any time since it is not referenced by the I/O daemon or by users. Also, the source segment can be recompiled at any time. Doing so changes the iod_tables segment, but not the iod_working_tables segment.

Each time the I/O coordinator is initialized, it compares the compilation time of iod_tables to that of iod_working_tables. If the compilation time of iod_tables is more recent, indicating that it has been recompiled, then the contents of iod_working_tables are replaced by the contents of iod_tables. Similarly, if no iod_working_tables segment exists (as would be the case at a new site), one is created with the contents of the iod_tables segment. Hence, changes to the I/O daemon tables do not take effect until the next I/O coordinator initialization. If an immediate change is necessary, then the coordinator must be logged out and logged in again.

At times it may become necessary to examine the contents of iod_tables, iod_working_tables, or some other object segment produced by the iod_tables_compiler. For example, one might suspect that the iod_working_tables segment has been damaged or one might lose the source segment from which iod_tables was generated. The print_iod_tables command (see Appendix A) essentially performs the inverse translation of that performed by iod_tables_compiler. Given any object segment generated by the iod_tables_compiler, print_iod_tables prints a source language description of that object segment. In fact, if the output from this command is directed to a segment, the segment can be compiled by the iod_tables_compiler to reproduce the object segment.

CREATION AND MAINTENANCE OF I/O DAEMON QUEUES

The I/O daemon queues are created automatically by use of the create_daemon_queues command (see Appendix A). The queues are created in the same directory as the I/O daemon tables, i.e., >daemon_dir_dir>io_daemon_dir. The command determines what queues to create based on information contained in the iod_tables segment. For each request type, one to four queues are created depending on the value of Max_queues or the per request type max_queues, whichever is in effect. The name of each queue is of the form XXX_N.ms where XXX is the request type name and N is the priority number. The ms suffix indicates that each queue is a ring 1 message segment.

Because the I/O daemon queues are message segments, access to the queues is determined by extended access modes. The IO.SysDaemon identity is given full extended access, i.e., add, delete, read, own, and status (adros) to all queues. For standard system queues (i.e., queues for which the driver_userid of the corresponding request type is IO.SysDaemon) aros permission is given to all users. Otherwise, the assumption is made that the queues are dedicated to the particular project named in the driver_userid. In this case, aros permission is given just to users of that project. The ms_list_acl command can be used to list the extended access on the queues and the ms_set_acl command can be used to change the extended access on the queues. All message segment commands are documented in the MAM System.

Changes to the I/O daemon tables must sometimes be coordinated with changes to the I/O daemon queues. In particular, when a new request type is added, new queues must be created for this request type. This can be done as soon as the iod_tables segment has been recompiled. Use of the create_daemon_queues command does not affect any existing queues, but does create new queues for any newly defined request types. If a request type is removed from the I/O daemon tables, the queues are not automatically deleted. The ms_delete command can be used to delete obsolete queues.

MAINTENANCE OF AIM FEATURES

At sites using authorizations above system_low, a special awareness is required of the way in which AIM affects the I/O daemon. To begin with, the I/O coordinator should always be logged in at system_high authorization. This is appropriate because the coordinator must distribute requests of all access classes. A driver process, on the other hand, does not necessarily process requests of all access classes. A driver is associated with a device that in turn is associated with a device class. The max_access_class for the device class defines an upper limit on request access classes handled by the driver. Hence, a driver authorization need be no higher than the associated max_access_class.

The access class of the io_daemon_dir directory must be system_low so that users of all authorizations have access to its various data bases. The iod_tables and iod_working_tables segments, for example, both have a system_low access class. This implies, of course, that the iod_tables segment can only be compiled at system_low authorization. The I/O daemon queues should have a system_high access class. This is possible because the queues are message segments which, unlike ordinary segments, can have a higher access class than their containing directory. The access class of a message segment is determined by the maximum authorization of the process that creates it. This implies that the create_daemon_queues command should only be used by persons having a system_high maximum authorization. Furthermore, because the queues are created in a system_low directory, the user of create_daemon_queues must have a system_low authorization.

The directories contained in the io_daemon_dir directory are potentially "upgraded," i.e., they may have access classes higher than that of io_daemon_dir. Specifically, the access class of the coord_dir directory equals the coordinator authorization while the access class of a driver directory equals the authorization of the corresponding driver. Thus, at sites using authorizations above system_low, upgraded subdirectories are created in io_daemon_dir. This implies that io_daemon_dir must have a quota so that quota can be moved to upgraded subdirectories (as required by AIM). The coord_dir directory, if upgraded, is assigned a quota of 250 records. Each driver directory is assigned a quota of 2 records if no minor devices are defined, or else 2 records per minor device. The quota initially assigned to the io_daemon_dir directory must take into account the requirements of these subdirectories plus the I/O daemon queues and the other segments in io_daemon_dir. Somewhere between 300 and 350 records is usually sufficient.

REQUEST TYPE INFO SEGMENTS

Each printer request type may have an optional request type info segment (rqti segment) associated with it that defines the physical paper characteristics, the logical VFU channel stops, and some additional driver control data. It is recommended that a special form have a specific request type and thus a separate set of channel stops. The channel stops are set only during driver initialization and remain constant for all requests done by the driver.

In addition, a site may wish to use the request type feature to group requests that use the same VFU tape, regardless of what preprinted form stock is needed for the request. By using the "^auto_print" driver mode, the operator may run requests associated with a given VFU tape (request type) in sequence and change the form stock on the printer to meet the needs of each request.

Printers that have firmware loadable VFC images are loaded by the driver during driver initialization (the paper may have to be realigned by the operator). For printers that use punched paper VFU tapes, the physical VFU tape for the request type must be mounted on the printer at the time the driver is initialized. The driver indicates the number of lines-per-page and the lines-per-inch switch setting that the operator should use.

The size of the head and tail sheets is set automatically to the physical dimensions of the paper as defined in the request type info segment.

The directory named >daemon_dir_dir>io_daemon_dir>rqt_info_segs must give sma access to the administrator and s to all other users. The initial ACL for segments must be set to rw for the administrator and r to all other users. AIM access, for those sites using the access isolation mechanism, should be system_low (the default).

This directory contains all request type info segments. If a single segment describes the paper characteristics for more than one request type, added names may be used in place of separate identical segments. Info segments are only required for printer request types that have the rqti_seg substatement in the iod_tables. When no rqti segment is used, the defaults described for the cv_prt_rqti command are used (see "Syntax for the Request Type Info Source Segment" below).

The printer rqti segments are created by the cv_prt_rqti table conversion command. The cv_prt_rqti command description appears in Appendix!A. A sample source file is shown in "Example of a Request Type Info Segment" below.

The contents of an rqti segment may be printed by the display_prt_rqti command. This command formats its output so that when directed to a file, the file can be used as input to the cv_prt_rqti command.

Syntax For The Request Type Info Source Segment

The request type info source segment contains keywords that define certain values put into the request type info segment. The general syntax is of the form:

keyword: <value>;

where the keyword defines a parameter to be set, and the <value> defines what the value of the parameter is.

The keywords and a description of the values acceptable to the cv_prt_rqti command are defined as follows:

driver_attributes: [^]value{,[^]value...};

The driver_attributes keyword is used to establish some operating parameters for the driver. There are two values defined: auto_go and meter. Each value may be preceded by the character "^" to negate the parameter. The driver_attributes keyword is optional (the default is ^auto_go,^meter).

The auto_go value is used to make the central site or remote printer driver request service from the coordinator immediately after initialization without asking for a go command.

For printers on remote stations that are always made ready to accept print files (e.g., where another computer simulates an RJE station), the auto_go value is particularly useful as a means of starting or resuming the processing of print requests without operator intervention.

The meter value is used to tell the driver to maintain internal metering data about its operation. (Note: metering is done according to the driver module design and not all driver modules implement metering.)

driver_wait_time: <number>;

The driver_wait_time keyword is optional (the default is 30 seconds.) It is used to set the time interval that the driver will sleep if there are no more requests in the queues. At the end of the interval, the driver will again ask the coordinator to check the queues for requests. The value is a decimal number between 30 and 300 seconds.

banner_type: standard | brief | none;

The banner_type keyword is optional (the default is "standard"). This keyword specifies to the driver whether the standard head/tail sheets will be printed for each copy of a request, a brief version, or none (separator bars only). The value must be either "standard", "brief", or "none".

banner_bars: double | single | none;

The banner_bars keyword is optional (the default is "double"). This keyword specifies to the driver how the separator bars at the bottom of the head sheet are to be printed. "Double" means overstruck separator bars, "single" are non-overstruck bars, and "none" causes the bars to be suppressed.

prt_control: [^]value{,[^]value...};

The prt_control keyword is used to set some driver request processing modes. There are four values defined: auto_print, force_esc, force_nep, and force_ctl_char. Each value may be preceded by the character "^" to negate its value. The prt_control keyword is optional (the defaults are auto_print, ^force_esc, ^force_nep, and ^force_ctl_char).

auto_print

This mode causes the driver to start printing each request as soon as it is received from the coordinator (after a go command has been given). This is the normal mode of operation. When this mode is turned off (^auto_print), the driver goes to request command level immediately after printing the log message. This allows the operator to align the paper, change the paper, print sample pages and issue all other commands allowed at request command level (including the kill command).

force_esc

This mode turns on the esc mode of the printer DIM during the processing of each request. This mode must be on if the slew-to-channel functions are to operate. (Note: users cannot set this mode from the dprint command.)

force_nep

This sets the noendpage (nep) mode of the printer DIM during the processing of each request, whether the user has requested that mode or not. This mode should be used for any request type that uses preprinted or preformatted paper (e.g., gummed labels, invoice forms, etc.) This causes the request to be properly formatted even though the user may forget to give the "-nep" control argument to the dprint command.

force_ctl_char

This sets the ctl_char mode of the printer DIM during the processing of each request, which allows an I/O daemon to send control sequences directly to a remote printer instead of discarding the characters or printing their octal equivalents. Setting this mode enables users who prepare print files through Compose to activate special printer features such as superscripting or multiple fonts. This mode is honored only by the remote printer driver module, remote_driver_.

message: <"string">;

The message keyword is optional. If specified, the value must be a character string enclosed in quotes, and may include newline characters. This character string must not be longer than 256 characters.

Any defined message is displayed on the operator terminal during the initialization of an I/O daemon driver for this request type. Typically, this message would tell the operator to mount some special form stock or which VFU tape number to use for this request type.

This page intentionally left blank.

message: <"string">;

The message keyword is optional. If specified, the value must be a character string enclosed in quotes, and may include newline characters. This character string must not be longer than 256 characters.

Any defined message is displayed on the operator terminal during the initialization of an I/O daemon driver for this request type. Typically, this message would tell the operator to mount some special form stock or which VFU tape number to use for this request type.

This page intentionally left blank.

paper_length: <number>;

The paper_length keyword is optional (the default is 66.) The value is a decimal number between 10 and 127 which specifies the number of lines on one physical page of the paper. The number of lines depends on the number of lines per inch that is used (see the "lines_per_inch" keyword). This number includes all lines, even though they may normally be used for top or bottom margins. For example, there are 66 lines on an 11-inch page at six lines per inch.

paper_width: <number>;

The paper_width keyword is optional (the default is 136.) The value is a positive decimal number that specifies the maximum number of character positions on one printed line. A warning message is given if a value greater than 136 is specified.

lines_per_inch: <number>;

The lines_per_inch keyword is optional (the default is 6.) The value is a number that specifies the vertical spacing used by the printer for this request type. The value must be 6 or 8.

line(<line_no>): <ch_1,ch_2,ch_3,...,ch_n>;

The line keyword is optional. There may be one line keyword for each line from 1 to the paper_length. The line keyword specifies which logical VFU channels are defined to stop at <line_no>. There may be 1 to 16 channel stops for any given line, each ch_i is a number between 1 and 16.

For example:

```
line(20):   1,5,11;
```

specifies that a slew to channels 1, 5, or 11 causes the printer to stop at the beginning of line 20.

NOTE: Line 1 is always defined as the form feed position. Typically the operator positions line 1 at the fourth printable line on a page.

end;

This keyword is required. The end keyword has no value. It specifies the end of the request type info source segment.

Example of a Request Type Info Source Segment

```
/* SAMPLE SOURCE FILE FOR A PRINTER REQUEST TYPE INFO SEGMENT  */
/* Source file:    invoices.rqti                               */
/* Data segment:   invoices                                    */

/* The first two keywords apply to the header data only.  */

driver_attributes:  ^auto_go;  /* the default */
driver_wait_time:   30;     /* number of seconds driver will   */
                            /* wait before asking coord again  */

/* The following keywords apply only to the printer_driver_    */

banner_type:         standard; /* normal head/tail sheets */
                               /* otherwise say "brief" or "none" */

banner_bars          double; /* overstruck separator bars */
                             /* can be "single" or "none" */

prt_control:         auto_print, ^force_nep, ^force_esc;

/* Message to the operator during driver initialization      */

message:
"For the invoices, use VFU tape number 12.
The form stock is in storage bins 22, 23, and 24.";


/* Physical Paper Info */

/* The form stock is only 80 print positions wide and
   72 lines per page at 8 lines per inch                    */

paper_width:         80;       /* default is 136 */
paper_length:        72;       /* default is 66 */
lines_per_inch:      8;        /* default is 6 */


/* Channel Stops */

/* The logical channel stops are defined as follows:        */

line(1):             1;   /* channel 1 is top of form */
line(3):             4;   /* chan 4 is the address line */
line(12):            7;   /* chan 7 is the first entry line */
line(60):            7;   /* and is also the bottom line */


end;
```

SECTION 3

OPERATION OF THE I/O DAEMON

The following material describes all of the capabilities of the I/O daemon and all of the commands and operating procedures needed to make use of these capabilities. In practice, at most sites, the commands needed for normal I/O daemon operation are contained in the exec_com segments, system_start_up.ec and admin.ec, and they need not be typed by the operator. However, the operator must become familiar with the material in this section so he can handle special requests and other unusual circumstances correctly.

All I/O Daemons (coordinator and drivers) use the iod_overseer_ process overseer. This process overseer should be specified for each daemon in the PMF of the daemon's project. Additionally, the PMF entry should specify the "^vinitproc" attribute in addition to this overseer. (See MAM Project for description of the PMF.)

I/O Daemons set their search rules differently from ordinary users. Instead of using the "default" set of search rules from the system search rules, they use the "io_daemon" set. These can be changed by the use of the set_system_search_rules command in the system_start_up.ec. (See MAM System.)

I/O Daemons running in test mode (via the test_io_daemon command) do not change their search rules. The search rules in effect at the beginning of the test remain in force.

LOGIN AND INITIALIZATION OF THE I/O COORDINATOR

The coordinator should be logged in before any drivers are created. At some sites the coordinator and drivers are all logged in automatically at system startup time. At other sites, the operator may be required to log in the coordinator from an ordinary terminal. To do this, the operator types:

```
login IO SysDaemon
```

The system replies:

```
Password:
```

and the operator types the password for IO.SysDaemon.

At most sites, however, the I/O coordinator is logged in as a consoleless daemon from the initializer terminal. In this case the operator types:

```
login IO SysDaemon source_id
```

where source_id is the message coordinator source name for the I/O coordinator.

This page intentionally left blank.

When the capabilities of the access isolation mechanism (AIM) are being utilized at an installation, the operator may have to specify the authorization he wishes for the coordinator (if the default authorization for IO.SysDaemon is not acceptable). An authorization is specified by an additional argument to the login command. See the description of the login command in the MPM Commands for details. For example:

```
login IO SysDaemon -auth <desired_authorization>
```

or:

```
login IO SysDaemon source_id -auth <desired_authorization>
```

This is the same as above except for the -auth login control argument. The driver is logged in at the authorization of desired_authorization. The actual names to be used to specify the desired authorization are defined by the system administrator.

Once the IO.SysDaemon is logged in, the system asks for an operator command as follows:

    Enter command: coordinator or driver

The operator responds by issuing the coordinator command (short form "coord"). He types:

    coordinator

If another coordinator has already been logged in, the following message is typed:

    iod_overseer_: Coordinator is already running.

At this point, the operator is again asked for a command. Only one coordinator is permitted.

The coordinator process next prints its version number and attempts to initialize itself. The first step of this initialization is to finish work not completed by the last coordinator process. If this step cannot be performed, the following message is printed:

    iodc_init: Warning--Cannot get old saved list.
    Some deletions may not be performed.

This error message is not fatal and coordinator initialization continues. However, if any other errors are encountered, the coordinator prints an error message followed by the line:

    Process cannot be initialized.

This indicates a fatal error that should be brought to the attention of the system administrator or other responsible person. When the initialization is finished, the message:

    I/O Coordinator initialized

is returned. At this point, the coordinator is at command level and ready to accept drivers.

COMMUNICATING WITH THE COORDINATOR

The coordinator performs its job automatically without requiring any instructions from the operator. Therefore, it is rarely necessary for the operator to communicate with the coordinator. Occasionally, however, the operator may wish to issue one of the commands described below under "Coordinator Commands."

INTERRUPTING THE COORDINATOR

It is never necessary to interrupt the coordinator in the course of normal operation. However, in the event of a coordinator malfunction, or other unusual situation, it is possible to send a quit signal to the coordinator. This signal causes the coordinator to suspend its communication with drivers and thus eventually bring all drivers to a standstill. For this reason, the quit signal should not normally be used.

The method for sending a quit signal to the coordinator depends upon the coordinator terminal. If the coordinator is logged in from an ordinary terminal, the operator should simply press the proper key to issue a quit signal

(e.g., ATTN or INTERRUPT). If the coordinator is logged in from the initializer terminal as a consoleless daemon, the operator types:

    quit source_id

where source_id is the source name for the coordinator. The coordinator acknowledges the quit signal with the message:

    "QUIT" received.

The coordinator then comes to command level and prints:

    Enter command.

At this point, any of the commands described below under "Coordinator Commands" can be typed by the operator with the exception of the term command. After each command is processed, the coordinator returns to command level and again prints:

    Enter command.

The operator should not send a second quit signal to the coordinator at this time. If a second quit signal is received, the coordinator ignores it and points out the mistake by printing the message:

    io_coordinator: QUIT already pending.

The coordinator should not be left in the quit state for an extended period of time since this effectively halts all active driver processes. The start command, described later in this section, is used to return the coordinator to normal operation following a quit signal.

COORDINATOR COMMANDS

The following is a list of coordinator commands.

1. logout

   logs out the coordinator. Normally, all driver processes should be logged out before the coordinator. If driver processes are not logged out, however, they automatically detect the fact that the coordinator has been logged out. The drivers reinitialize and wait for a new coordinator to be logged in.

2. list

   causes the coordinator to print a list of active devices, i.e., devices currently assigned to drivers. The request type and current request number are printed for each active device.

3. print_devices

   causes the coordinator to print a list of all devices managed by the I/O daemon. The devices are grouped according to the request types they service. An asterisk (*) appearing before a device indicates that the associated request type is the default for the device. The driver access name and the driver authorization (if any) are given for each request type.

4. wait_status

   causes the coordinator to print a list of device classes for which requests have been added to the wait list. The number of waiting requests for each of these device classes is also printed. Requests

are added to the wait list whenever a driver gives the "next" command, or if the coordinator finds a request for a device class that is not currently active. At sites having only one device class per request type, no requests are automatically added to the wait list. At sites having multiple device classes per request type, requests may be held waiting whenever one or more drivers are active for a request type. By examining how many requests are waiting for various device classes, the operator can judge when it is appropriate to switch a device from one device class to another so that all device classes receive adequate service.

5. term device_name

   terminates a driver so that the major device (and all minor devices) assigned to it can be assigned to another driver. The device_name for the driver must be specified following the command. Normally, driver termination is performed automatically when a driver logs out. In the case where a driver process terminates abnormally, the coordinator does not discover that the process is terminated until a new driver attempts to log in; then it is unassigned from the old driver process and is assigned to the new driver process. Therefore, the only time it is necessary to use the term command is when one wishes to terminate an active driver that cannot be logged out. This might be necessary, for example, if the driver is logged in from a remote location. (If the driver process is running, the term command will not cause the driver to detach the channel associated with the major device. It will cause the driver to eventually fault and probably destroy itself.)

6. restart_status

   causes the coordinator to print the number of restartable requests for each different request series and to identify those request series for which a restart cycle is in progress."

7. help

   list commands acceptable to the io_coordinator.

LOGIN AND INITIALIZATION OF DEVICE DRIVERS

Device drivers must be logged in after the coordinator. If there is no coordinator logged in, a driver waits up to 5 minutes for one to log in. Otherwise, it prints an error message and logs out.

Logging in a Driver

At some sites, the drivers are logged in automatically at system startup time. At other sites, the operator may be required to log in the driver. The normal login identifier for the driver is IO.SysDaemon, the same as for the coordinator. The procedure for logging in a driver is the same as that described earlier for the coordinator.

Driver Initialization

Once the driver has been logged in, the system asks for an operator command as follows:

Enter command: coordinator or driver

The operator responds by typing:

driver

At this point, if no coordinator is logged in, the driver waits for one (up to 5 minutes).

If the coordinator is present, the driver prints its version number and starts to initialize itself. Once the initial checks are complete, the driver prints:

Enter command or device/request_type:

The operator responds by typing a command or the name of a device. If the names of the devices are unfamiliar, the operator may use the print_devices command to get a list of devices and request types that may be used. (This is the same as the print_devices command described earlier under "Coordinator Commands.")

The operator may type "help" to get a list of acceptable commands. These are:

```
<device_name> {<request_type>}    runs the given device
listen <line_id>                  waits for remote station to dial the line
print_line_ids                    prints line_ids for the listen command
print_devices                     prints device names and request types
logout                            logs out the driver
```

If a device name without a request type is given, the default is used. A request type must be given when the operator wants the device to process a request type other than the default. For example:

```
   Enter command or device/request_type:
!  prtb unlined
```

allows the operator to process requests intended for printing on unlined paper (of course, the unlined paper must be loaded on device prtb).

When the device performs multiple functions, such as the case of a remote printer/punch, a minor device is specified for each logical function the device supports. Each minor device can process requests of different request types. When the default request type for each minor device is to be used, the operator should specify the request type keyword: "default". For example, when using a multifunction device called "xyz", the response might look like:

```
   Enter command or device/request_type
!  xyz default
```

NOTE: The default request type keyword is always accepted, even if there are no minor devices defined for the device specified.

When the operator wishes to prepare the driver to wait for a remote station to log in, he responds with "listen <line_id>". The driver will look up the logical line_id in the iod_working_tables and attach the line as specified by the attach description. For example:

```
   Enter command or device/request_type;
!  listen 2780_1
   Attaching line "2780_1" on channel (a.h001)
```

The driver will remain blocked until a device dials up on the specified line. Then the driver will respond:

Requesting station identifier on line "2780_1".

and the operator of the device which dialed up will be asked to enter the station command:

station station_name password

The driver validates that station_name is registered with the specified password and then initializes the driver using the device called station_name and the default request types (if defined) for each minor device.

When the default request type(s) is to be used and no default is defined (or if the request type specification is ambiguous), the operator will be asked to specify the request type to use for each minor device. For example, in the case of the multifunction device "xyz":

```
   Enter command or device/request_type
!  xyz

   Enter request type (or default) for minor device "print":
!  xyz_prt

   Enter request type (or default) for minor device "punch":
!  xyz_pun
```

The operator may type "new_device" if he wants to abort initialization of the major device xyz and initialize a new major device. The default keyword may also be used here to specify the default request type for a minor device.

If there is some reason why the driver is not allowed to operate the specified device(s) or request type(s), one or more error messages are typed explaining the problem and the driver again asks for a command or device/request_type.

Once the device(s) and request type(s) are known, the driver communicates with the I/O coordinator to announce its readiness. If the coordinator refuses to accept the new driver, a message is printed and the operator is asked to enter a new device name.

When the coordinator has accepted the driver, the driver prints any special initialization messages for the device. After all initialization has been completed, the driver responds by printing:

<device> driver ready at <time>

Enter command:

The driver is now at command level and is ready to start processing requests. The operator enters any commands needed to cause the driver to operate as desired. Finally, the operator enters the go command to begin processing requests.

Extensive error checking is performed during initialization. Certain errors are fatal to initialization and others are not. When an error occurs that is not fatal, a message is typed on the terminal and the driver requests operator instructions. Fatal errors are denoted by the words "Fatal error:" in the error message followed by a logout of the driver or a request for a new device. If a fatal error occurs, an administrator or member of the programming staff should be notified prior to any further action by the operator.

Driver Initialization When Using Device Classes

A device class is a subdivision of a request type. When the administrator has specified that a request type has one or more device classes, the operator must identify the device class as well as the request type he wishes to use with a device. The print_devices command provides the needed information.

The driver should be logged in at an authorization equal to the max_access_class specified (by the system administrator) for the device class. If the driver authorization is incorrect, an error message is printed and the operator is asked for a new device and request type.

If the default request type and device class are desired, no action is needed other than that described above. However, if a different request type or device class is to be used, the operator's response must be slightly different. The optional request type must be entered in the form:

<request_type>.<device_class>

to indicate that <device_class> of <request_type> is to be used.

For example, suppose there are two device classes defined for request type printer: general and restricted. When the operator wishes to use the general device class on prta and the restricted device class on prtb he would do the following:

1. log in the first driver and get the message:

   Enter command or device/request_type:

   and respond by typing:

   printer_1 printer.public

2. log in the second driver and get the message:

   Enter command or device/request_type:

   and respond by typing:

   printer_2 printer.confidential

Now all user requests for request type "printer" in the access class range specified for device class "public" are processed on printer_1. Similarly, all user requests for request type "printer" in the access class range specified for device class "confidential" are processed on printer_2.

If this were the normal mode of operation, the default types for printer_1 and printer_2 could be specified as printer.public and printer.confidential, respectively. Then the operator could omit the request type and device class, as described earlier.

TERMINALS THAT CONTROL THE DRIVER

A driver process is capable of receiving commands from two sources: the normal login terminal (master terminal) and a slave terminal. The driver MUST have a master terminal, but the slave terminal is optional. For most devices, the slave terminal is an additional terminal attached to the driver. This is also called the control terminal. Any driver can have a control terminal specified (but it is meaningless for some drivers, e.g., the spool driver). For devices that have a multifunction device, the device itself can act as a slave terminal.

When the site administrator has specified that a control terminal is to be used with a device, the driver is not able to complete its initialization until the control terminal has been attached, except for remote stations.

The slave terminal functions as a source of driver commands and a place to write error messages, operational messages, and log messages.

The control terminal is primarily used to prepare receipts or accountability forms to control the distribution of output. The control terminal will take on the functions of a slave terminal if there is no other slave terminal defined.

The control terminal is not always a slave terminal for a remote station. A Type I station must login with the device providing command input. Hence, the device will remain as the slave terminal, even though a control terminal may be specified for the device. A Type II station may use a control terminal as a slave terminal. Also, by typing "slave= yes" in the device args string of the iod_tables, the device may become the slave terminal even though a control terminal is already attached.

Master Versus Slave Functions

The authority of the master terminal over the slave terminal ensures that central operations has full control of the driver at all times. The slave terminal is provided for decentralization of operational control when this feature is needed. When the driver has a slave terminal, most operational messages, such as requests for commands, are sent to the slave terminal instead of the master terminal.

The master terminal is assured control at command level by not allowing a quit signal to be issued from the slave terminal while the master terminal is executing a command. Also, the master terminal can hold the driver at command level indefinitely if necessary. (See "Standard Driver Commands" later in this section.)

Otherwise, the slave terminal can perform almost every function that the master terminal can. When the slave terminal is a control terminal, a short message is printed every time a request is processed. This message can be reformatted at the site to provide a more formal accounting for output generated by a driver. (See "Using Preprinted Accountability Forms on the Control Terminal" later in this section.)

Driver Initialization With A Control Terminal

When a control terminal is to be attached by the driver, there is an additional step performed just before the driver comes to command level. The driver waits for the control terminal to be assigned to it by the system control process. The driver may request a specific terminal to be assigned or it may wait for a terminal to "dial" the driver process. Normally, no action is required by the central site operator. Only the control terminal operator is allowed to take action to connect the terminal. The sequence of messages might look like this on the master terminal:

```
     Enter command or device/request_type:
  !  prt_x
     prt_x driver waiting for control terminal "<dial_id>" to dial.
```

After the control terminal operator "dials" in the control terminal, the driver continues with:

```
   Control terminal accepted.

   prt_x driver ready at 02/02/78 0200.0 est Thur
```

Now the driver is at command level.

While the driver is waiting for the control terminal, the control terminal operator must dial the terminal to the driver. First the terminal phone connection must be completed. After the normal greeting message, instead of using the login command, the operator types the dial command:

```
     dial <dial_id> <driver_userid>
```

where <dial_id> is the identifier specified in the driver message above, and <driver_userid> is the login identifier of the driver (normally IO.SysDaemon). The control terminal operator must know the <dial_id> for the particular driver. However, this is not protected like a password. It only serves to distinguish between the various driver processes with the same <driver_userid>.

When the dial command is accepted, a connection message followed by the driver's ready message is printed on the control terminal:

```
     prt_x driver ready at 02/02/78 0800.0 est Thur

     Enter command:
```

At this point commands are accepted from either the master or slave (control) terminals.

DRIVER COMMAND LEVELS

The driver supports several different command levels, each associated with the function to be performed. These are not the normal Multics process command levels and only limited sets of commands specific to a driver process are accepted. Each command level other than the normal driver command level identifies the function by a word in parentheses following the command request (e.g., "Enter command(quit):" for quit command level). Not all commands may be used at every command level. The operator should be aware of any command level restrictions identified in the command descriptions below.

Normal Driver Command Level

A driver process indicates that it is at normal driver command level by printing the request:

```
Enter command:
```

The driver comes to normal command level as soon as all initialization is complete and also after each request is finished, if commands have been given or step mode has been set.

Request Command Level

Request command level is used by some device drivers to allow the operator to modify the normal processing of a request. For example, the printer driver uses the request command level to allow the operator to specify the starting page of the request or print sample pages for alignment.

The printer driver comes to request command level when it is running in ^auto_print mode. The remote driver comes to request command level when it is running in ^auto_print mode or ^auto_punch mode. (This mode is set by the prt_control command or the rqti segment initial value.) Request command level is indicated by the driver typing:

```
Enter command(request):
```

Request command level is distinguished from normal command level by the word "request" in parentheses. The use of request command level is a device specific function. For a list of commands issued from request command level, the user should refer to "Device Specific Driver Commands" later in this section. Most other standard driver commands are also available at request command level.

Quit Command Level

A quit signal is transmitted to the driver in a manner similar to the way it is transmitted to the coordinator (as described earlier). When a quit signal is received, the driver suspends its current operation and comes to quit command level as indicated by typing:

```
* QUIT *
Enter command(quit):
```

Quit command level is distinguished from normal command level by the word "quit" in parentheses.

Several standard driver commands can only be used at quit command level; they are described below. Most driver specific commands may also be used at quit command level.

STANDARD DRIVER COMMANDS

The two classes of commands for a driver are: standard driver commands and device specific driver commands. The device specific driver commands are described later in this section under "Device Specific Driver Commands." The standard driver commands are described here, grouped by their function. Detailed descriptions of all the driver commands are given in alphabetical order at the end of this section.

1. general control

ready	makes device ready to process requests
go	begins processing of requests
halt	stops processing of requests on a device
logout	causes driver to log out, except for remote drivers
step	causes driver to wait after each request
hold	holds the driver at command level
new_device	causes driver to request new device
inactive_limit	sets time limit for inactivity logout
auto_start_delay	sets wait time between quit signal and start command
defer_time	sets time limit for automatically deferring requests
x	executes site-defined exec_coms

2. control after interrupting a request (quit)

start	resumes driver operation
kill	terminates the current request
cancel	terminates and discards the current request
restart	causes reprocessing of the current request
defer	sends current request back to its queue
save	saves the current request for possible restarting

3. information

help	lists all driver commands
status	lists current status of the driver

4. coordinator communication

restart N	causes reprocessing of previous requests
save N	saves requests for possible restarting
restart_q	returns to the head of each queue
next	runs a specified request next

5. terminal control

slave_term	controls use of a slave terminal
ctl_term	controls operation of a control terminal
slave	sends a message to the slave terminal
master	sends a message to the master terminal

6. error recovery

reinit	reinitializes the driver
release	returns driver to normal command level

Some commands perform more than one function. However, these are clearly distinguished by control arguments.

General Control Commands

General control commands (item 1 above) are used at normal command level to initiate and control the operation of the driver. This set of commands is sufficient to run the driver if no unusual circumstances are encountered.

Control Commands After Interrupting a Request

A request is interrupted by giving the driver a quit signal. The commands shown in item 2 above can only be used at quit command level, although two have other uses in different contexts. These commands are useful for modifying the driver's sequence of operations:

cancel	kill	restart
defer	logout	save
halt	new_device	start
help	ready	status
hold	release	step

If no commands have been given to the driver within 60 seconds following a quit signal, an automatic start command is executed by the driver. The 60 second delay can be adjusted by the auto_start_delay command.

Information Commands

The commands in item 3 above provided to furnish additional information to the operator. For device specific driver commands, the help command identifies those commands that may be used for a given driver.

Coordinator Communication Commands

These commands (item 4 above) are used by the operator to instruct the coordinator in how to handle requests. The operator must be able to prevent the loss of requests due to device malfunction. To this end, the coordinator retains each completed request in a "saved" list for a period of time to allow each one to be reprocessed if needed. The operator is able to shift the priority of individual requests.

The coordinator keeps track of the requests in the list by their request numbers. The request number argument to the save and restart commands is used to identify requests to the coordinator. A request number is composed of a request series and a sequential number indicating the order in which the request was processed. For example, request number 50289 is the 289th request processed by the device within the 50000 request number series. Each device or minor device is assigned a series of 10000 sequence numbers during initialization. The first series after coordinator initialization begins at 10001, the second series begins at 20001, and so on. This ensures that each request in the coordinator's "saved" list is uniquely identified.

Commands For Terminal Control

To ensure the master terminal's ability to define the functions of the slave terminal, two commands are provided (item 5 above) to control how the driver treats slave terminal input and output.

Since the slave terminal can be the device itself or an additional terminal, the functions that allow the site operator (or device operator) to control the slave are separated into two commands: 1) those applying to all slave terminals (the slave_term command), and 2) those that only apply to an additional control terminal attached to the process (the ctl_term command).

Error Recovery Commands

The commands in item 6 above are provided for error recovery. There may be circumstances that make the driver unable to continue its operation. This could occur if the coordinator process were terminated or if some control data were

destroyed. When the driver can identify the problem, it takes some action, if possible, to correct the situation.

Under some conditions, it may be necessary for the operator to reinitialize the driver or even to log out without completing any pending requests.

DEVICE SPECIFIC DRIVER COMMANDS

The device specific driver commands allow the operator to control the operation of different devices. Each driver module is capable of implementing any commands necessary to control the operation of its device.

Driver modules are designed to be molded by a site into a form necessary to support its own devices, each with its own set of commands. The operator should familiarize himself with the commands associated with the driver modules used at an individual site.

Standard device drivers operate printers and punches and can read card decks from remote multifunction devices. One driver even writes printer requests onto tape (spool_driver_). There are different device specific commands for these generic functions and some additional commands associated with operation of physical devices. The operator can use the help command of the driver to display the full set of commands the specific driver can accept. The device specific commands implemented by standard device drivers are listed as follows:

1. commands for printers:

banner_bars	defines printing of separator bars
banner_type	defines what is printed on the banner
paper_info	defines paper length, width, and lines per inch
prt_control	defines printing control functions
sample_hs	prints a sample head sheet banner
single	single spaces on formfeed and vertical tab

2. commands for printers (request command level only)

copy	sets the copy number of the next copy
print	prints the next copy starting at the current page
req_status	gives status info about current request
sample	prints a sample of the current page

3. commands for local punches

   (no special punch device commands are required; standard command may be used)

4. commands for remote punches

pun_control	sets the punch control modes (does not apply to the central site punch driver)
sep_cards	controls punching of separator cards between each output deck

5. command for remote punches (request command level only)

copy	sets the copy number of the next copy
punch	punches the next copy of current request
req_status	gives status info about current request

6. commands for card input

clean_pool	deletes old card decks
read_cards	starts card input

7. command for control of terminal operation (most drivers)

sample_form	prints a sample control form

8. commands for remote device control

pause_time	sets pause time between requests
runout_spacing	sets paper advance after a command request

9. commands for the spool driver

banner_bars	defines printing of separator bars
paper_info	defines paper length, width, and lines per inch
prt_control	defines printing control functions
sample_hs	prints a sample head sheet banner
single	single spaces on formfeed and vertical tab

MAKING THE DRIVER ASK FOR A COMMAND

A command may be entered from the master or slave terminals at any time after the driver has been initialized. However, when requests are being processed continuously, the messages printed on the terminal may interfere with operator input. Therefore, it is better to make the driver ask for a command and wait for the operator to respond. This may be done in two ways:

1. During a pause in terminal printing, the operator may simply press the newline key of the terminal. This causes the driver to ask for a command before processing the next request. (A go command is required to allow the driver to continue.) When using the message coordinator, the operator should send the hold command to the driver process.

2. At any time the operator may issue a quit signal to the driver. This suspends the current request while the driver asks for a command. After a quit signal, if the operator wishes the driver to finish the current request and return to command level, he may give the step command followed by the start command.

NOTE: When the driver is simulating form feeds on the control terminal, a quit signal terminates form alignment. The driver completes the control terminal message, if possible, before asking for a command. However, the ctl_term aligned command or the sample_form command must be given before the driver can accept a start command. (See "Using Preprinted Accountability Forms on the Control Terminal" later in this section for a more detailed explanation of these commands.)

ENTERING COMMANDS FROM A MULTIFUNCTION DEVICE CARD READER

A card reader in certain multifunction devices can be used as a slave terminal to input commands. Driver commands must be punched on cards, one command line to a card.

USING PREPRINTED ACCOUNTABILITY FORMS ON THE CONTROL TERMINAL

A control terminal may be used to produce accountability records which correspond on a one to one basis with each copy of each request processed by the driver. The format of the accountability record may be redefined by each site for each driver. In some security related applications, this feature may be used to fill in the blanks on preprinted document accountability forms to provide a record of each piece of output.

The format of preprinted forms is likely to be different at each Multics site. The administrator must ensure that a program is provided to correctly print the request data on each form. He also must specify the form type identifier to be used with the ctl_term command to establish the site program for printing forms. (The default form type for all drivers is a one-line message per request.)

The operator must use the ctl_term form_type command to change the control terminal message to the desired format. Once the new form type has been accepted by a driver, the operator should use the sample_form command to ensure correct alignment of the data on the form. This is normally all that is needed as long as the terminal hardware provides a form feed capability.

Otherwise, the operator uses the form feed simulation functions of the driver to ensure continued alignment. It is very important that the dimensions of the form be specified by the driver command. (The commands that set form feed simulation and form size can be part of the exec_com that initializes the driver.)

The following is an example of the command sequence to simulate form control of a preprinted form with dimensions 8 inches wide by 5 inches long. (Operator input is denoted by the exclamation point (!).)

```
     Enter command:
  !  ctl_term  simulate
     Forms will have to be aligned.
     Enter command:
  !  ctl_term page_length  30
     Enter command:
  !  ctl_term modes 1170
     Enter command:
  !  sample_form
       .
     [sample data is printed on the control terminal]
       .
     Enter command:
  !  go
```

In this example, there are 30 lines to a 5-inch page and 70 characters to an 8-inch line (allowing for margins). The sample_form command should be given repeatedly until correct alignment of the form is achieved. Finally, the go command is given to begin processing requests.

When the operator issues a quit signal to the driver (from any terminal) or when an unknown command is entered from the control terminal, the form alignment is assumed to be incorrect. Therefore, the driver demands that the sample_form command (or the ctl_term aligned command) be given before the next go command (or start command after a quit signal). This situation might look like:

```
     Enter command:
  !  goo
     prta driver:  Invalid command for driver - goo
     Enter command:
  !  go
```

```
    Control forms not aligned.
    Enter command:
!   sample_form
     .
    [sample form is printed on the control terminal]
     .
    Enter command:
!   go
```

Now the driver can continue processing requests. Form alignment is ensured for all input and output on the control terminal as long as:

1. commands are entered only when requested
2. commands from the control terminal are entered correctly
3. no quit signals are issued
4. the control terminal maintains paper alignment on the platen

DEVICE SPECIFIC DRIVER OPERATIONS

Some drivers give special messages and/or request special instructions from the operator during initialization or during operation. The following paragraphs describe any special messages or operations for the standard device drivers.

Operation of the Printer Driver

LOGIN AND INITIALIZATION

The printer driver is logged in like all other drivers (as explained in "Login and Initialization of Device Drivers" above).

After the operator gives the response to the following:

```
Enter command or device/request_type:
```

any operator message defined in the rqti segment for the request type is printed on the operator terminal, e.g.:

```
For the invoices, use VFU tape number 12.
The form stock is in storage bins 22, 23, and 24.
```

Next, any paper characteristics defined in the rqti segment are given to the printer software. If the printer is capable of receiving a firmware VFC image from Multics, it is loaded and no further action by the operator is needed. However, if the printer uses punched VFU tapes, an additional message is printed on the operator terminal, having the form:

```
Mount VFU tape for 72 lines per page.
Set printer for 8 lines/inch.
```

The driver completes its initialization and waits at command level for the operator to ensure that the printer is ready:

```
printer_1 driver ready at 01/30/78 1405.2

Enter command:
```

After the correct VFU tape has been mounted on the printer, the driver is ready to start processing requests. The operator can use the command "sample_hs" to print a sample head sheet and verify the correct paper alignment. Request processing may be started in normal or step mode as desired by using the "go" command.

The auto_go driver attribute in the rqti segment may be used to make the driver skip the request for an initial command and immediately look for the first request to process. However, the auto_go is cancelled if the printer cannot accept a firmware VFC image.

LIMITATIONS

With a PRT1200 or PRT1600, 20-lb paper should be used. (The use of lighter weight paper may prove problematic.)

PROCESSING REQUESTS

As each request is received from the I/O coordinator, the printer driver prints a short description of the request in the log, similar to:

```
Request 10001 printer q3: >print_files>invoices>Station_A.invoices
  from Username.Project.a (for "Heading" at "Destination")
  Time estimate for request 10001: 12.3 minutes
```

The time estimate is shown only if the estimate exceeds 1 minute.

If a maximum request time limit has been set by the defer_limit command, and the estimated processing time of the request exceeds this limit, the message will look like:

```
Request 10001 printer q3: >print_files>invoices>Station_A.invoices
  from Username.Project.a (for "Heading" at "Destination")
  **Deferring Request 10001.  Printing time estimate: 12.3 minutes
```

If the driver is running in auto_print mode (the normal default), the request is printed immediately. When completed, a message is printed in the log, giving the charge for the request:

```
Charge for request 10001:  $36.20  (27546 lines,  1523 pages per copy)
```

The driver then asks the I/O coordinator for the next request (or returns to normal command level if in step mode).

If the driver is not running in auto_print mode, the driver comes to the request command level by printing:

```
Enter command(request):
```

This page intentionally left blank.

This is not the normal driver command level. The driver is now ready to accept additional request control commands (plus a help command) to specify the starting page, to print a sample page, or to set the copy number of the current copy. These request control commands are described above under "Device Specific Driver Commands." At this point, the operator should verify that the correct paper stock is on the printer, aligned at the top-inside-page position. The operator may verify the alignment of the paper by printing a sample of the starting page (specified by the operator) before printing the file.

After the "print" command is given, the driver prints a head banner and the text of the file from the starting page to the end of the file and completes the request as described above.

Operation of the Punch Driver

There are no special messages or commands for the punch driver. The Multics punch driver follows the general operation of I/O daemons described above. Log messages, time estimates, and charges shown for each request are similar to those of the printer driver. The punch driver does not implement request command level.

Operation of the Spool Driver

The Multics spool driver provides an alternative method for processing users' print requests when the service printer is either down or substantially backlogged. The spool driver obtains queued print requests from the coordinator and writes the requests out onto magnetic tape. The tape can then be processed immediately or at a later time in one of two ways: the spooling tape can be input to a Multics system using the print_spooling_tape command (described in Appendix A) to write directly on the printer, or the spooling tape can be input to another system that has software capable of reading and printing the contents of the tape. The spool driver does not implement request command level.

LOGIN AND INITIALIZATION

The spool driver runs as a standard I/O driver process and can process printer requests. If the printer request type queues are to be used by the spool driver while the printer driver is logged in and working, the following situation arises: two drivers of the same request type share the processing of requests from the same queues in a round robin fashion -- the first ready driver getting the next request in the queues. This scattering of print requests can result in the printing of two adjacent requests in the queue at significantly different times. To avoid this request scattering problem, the printer driver should be logged out (or placed in "hold") before bringing up the spool driver.

The spool driver is logged in like any other driver and its operation is selected by the operator when the driver requests:

    Enter command or device/request_type:

The operator responds by typing the name of the spooling device, and either gives a request type or relies on the default request type (specified in the iod_tables).

Any special initialization messages associated with the request type are printed at this point. The line length, page length, and lines per inch to be used in printing the tape contents are printed for the operator.

Next, the tape data must be entered. At least one tape volume identifier (tape number) must be supplied, but additional data may optionally be supplied. This optional input includes a recording density and a number of requests (files) or a number of lines to limit the spooling operations. The spool driver asks for this information by printing:

    Enter volids and optional tape data or limits:

For example, if the operator wishes to spool 95 print requests to volume 070064 to be recorded at 800 bpi, he types:

    -volid 070064 -density 800 -files 95

The operator may make a selection from the following possible input parameters:

-volid STRs, -vol STRs
where STR is a six-character volume identifier of a tape reel. Up to three volid (separated by spaces) may be specified at one time, and at least one volid must be specified.

-density N, -den N
where N is either 800 or 1600. If the -density control argument is not specified, and the -interchange control argument is not specified, the default density is 1600 bpi. Density can only be given once during a spooling session or an error is indicated.

-interchange, -int
specifies tape recording parameters that comply with the ANSI standard requirements for interchange. With this control argument, tape block size is set to 2048 characters and recording density is set to 800 bpi.

-files N, -fl N
where N is a number between 1 and 999999, indicating the number of files (requests) to be written to tape before stopping. There is no default file limit. If this parameter is omitted, no limit is set on the number of spooling requests.

-lines N, -ln N
where N is a number between 1 and 999999, indicating the number of printed lines to spool before stopping. There is no default line limit. If this parameter is omitted, no limit is set on the number of lines spooled.

When the coordinator accepts the spool driver as a driver and all the preliminaries of validating the input parameters have been completed, the spool driver prints:

```
Spool driver ready at 01/30/78  1452.8 edt Mon

Enter command:
```

The spool driver is now at normal command level and ready to start processing requests. At this point the operator can modify the paper printing parameters with the paper_info command if desired. All standard driver commands can be used as well as most device specific driver commands for printers. (The spool driver does not support a request command level.)

To begin processing requests, the operator must type the go command. Assuming that some outstanding print requests are queued, the spool driver starts processing requests at the go command. The first print request message is printed on the spool driver log, followed by a tape mount message; after the first tape reel has been mounted requests continue to be processed and logged sequentially until either the queues become empty or one of the spooling limits has been reached. The spool driver output log looks something like the following:

```
Request 10001 printer q3: >udd>Demo>JSmith>test.
  from JSmith.Demo.a (for "heading" at "destination")
  Charge for request 10001.3:  $1.65 (1055 lines,  10 pages)

Mounting volume xxxxxx with a write ring.
xxxxxx mounted on tape_04.
```

```
Request 10002 printer q3: >udd>Demo>js>test1
Request 10003 printer q3: >udd>Demo>js>test2
  .
  .
  .
  .
```

When any spooling limits have been reached, i.e., either lines limit or files limit, the spool driver prints:

```
Reached specified spooling limits;

        Current file limit is xxx
        Current line limit is xxx

        Current file count is xxx
        Current line count is xxx

Enter new file and/or line limits, or "detach":
```

The current file count is a tally of the number of files spooled so far. The current line count is a tally of the number of lines spooled so far. Current line limit is the line limit stop last set. Current file limit is the file limit stop last set. If the limits are zero, then they are not currently set. Each copy that a user requests corresponds to one file spooled, but the limits are approximate, as a request is processed completely before the limits are checked.

At this time, the operator must choose to either enter new spooling limits and continue, or to terminate the spool driver. If new line limits are specified, the new limit is added to the current limit, and spooling continues until that new limit is reached. If only a line limit is specified, the files limit is set to zero and only reaching the line limit halts spooling; and likewise, if only a file limit is specified, the line limit is set to zero and only reaching the file limit halts spooling. If both incremented limits are specified, they are both incremented and spooling continues until one of the two limits is reached, whichever one comes first.

TO CONTINUE SPOOLING

If the operator wishes to continue when the spooling limits have been reached, he must renew the limits by entering new -files and/or -lines parameters. The new values are added to the current spooling limits. For example, at this time the operator types:

```
-files 20 -lines 20000
```

to add 20 to the current file limit and 20000 to the current line limit and spooling continues.

If the end of a volume is reached when only one volume identifier has been specified, the spool driver asks for additional volume names:

```
Reached end of spooling volume list;
Enter more volids or "detach":
```

Here the operator types in another volume identifier, -volid STR, to continue spooling or types detach to terminate spooling.

TO TERMINATE SPOOLING

If the operator wishes to terminate spooling when spooling limits have been reached, he types "detach." The spool driver responds with a tally of files and lines processed and then logs out.

SPOOL DRIVER MESSAGES

The spool driver automatically answers all questions asked by the tape_ansi_ I/O module. The operator should not have to type answers to any questions from tape_ansi_ that appear in the spool driver log. For example, should a given volume need initialization, the following sequence of lines might appear on the spool driver terminal:

```
tape_ansi_: Volume xxxxxx requires initialization, but
cannot read VOL1 label.
Do you want to initialize it?  ! yes
```

SPOOL DRIVER COMMANDS

The special commands available to the spool driver are a subset of those listed for printers. The spool driver does not have a request command level. The commands available are:

banner_bars
paper_info
prt_control
sample_hs
single

Operation of a Remote Driver

The remote driver is designed to operate differently from most other drivers. The major difference is that it is designed for unattended operation at the central site. Normally, all commands will come from the remote terminal input device. However, the central site operator is always able to override the remote station operator. See the discussion of master and slave terminals earlier in this section.

At the central site, the operator starts initialization using the listen command. For example,

```
  Enter command or device/request_type:
! listen 2780_1
  Attaching line "2780_1" on channel (a.h001)
```

The driver will not wait for a remote station to dial in.

The following paragraphs concern the operation and control of the driver from the remote station.

INITIALIZING AND DIALING IN THE REMOTE STATION

The remote station operator must turn on the remote terminal and complete its initialization according to the manufacturer's instructions. Be sure the communications lines are connected to the modem and the terminal is configured to receive data from Multics. In particular, the terminal must be configured to transmit single card images if commands are to be read from cards.

The remote station operator must then complete the connection of the terminal to the I/O daemon. This is similar to the normal "login" of a process, except that the I/O daemon process is already logged in and waiting for the remote terminal to dial into the process.

The operator must dial the central site phone number that is to connect to the remote station. At this point the I/O daemon asks the remote station operator to:

    Enter station command:

The operator must then supply the station command through the card reader or the terminal, if it has a keyboard. (See the description of the station command later in this section.) The station command identifies the station_id and password assigned to the remote station as supplied by the system administrator. If the terminal has an operator's station with both keyboard and CRT or printer, the operator may omit the station password from the station command; the system will then request the station's password via the prompt:

    Enter station password:

and will either suppress printing of the password or print a mask to hide the password.

Once this information is validated, the driver will start initialization of the major device that has a name equivalent to the station_id. If there are default request types defined for any of the minor devices, these will be used. Otherwise, the remote operator will be asked:

    Enter request type for minor device <name>:

The operator may enter any request type that has been defined for the minor device in the iod_tables.

Remote stations that have no input device must use a dedicated communications line. In this case the driver is initialized from the central site as though it were a peripheral driver (i.e., the device/request type is given in place of the listen command). When the station dials into the dedicated communications line, no station command is requested. Instead, the driver will immediately begin initialization for the predefined device and request type associated with the driver.

The driver responds with any special rqti segment initialization messages or messages for setting the VFU tape and lines per inch on the printer minor device. The driver then responds with the message:

```
<device> driver on channel <channel_id> ready at <time>
Enter command:
```

The driver is now at normal command level and all the standard driver commands can be given from the remote station by placing cards in the reader (or input through some other terminal device).

If any printer minor devices specify an rqti segment that includes the auto_go driver attribute, those devices will automatically be readied as the driver initializes. The driver will then skip the request for an initial command and immediately look for requests to process for those minor devices.

The driver for a remote station provides several device-specific driver commands that control any reader, printers, or punches. These are issued from normal command level. After a request has been received from the coordinator, the driver can come to request command level. This is enabled by the ^autoprint mode for printers (see the prt_control command) and the ^autopunch mode for punches (see the pun_control command).

The driver can accept commands to alter the processing of the current request while at request command level. The commands are different for printers and punches.

At request command level, a printer device can be adjusted to a specific starting page or copy number and can print sample pages, as well as most other driver commands.

This page intentionally left blank.

At request command level, a punch device can adjust its copy number. The basic use of request command level for a punch is to make the driver pause after printing the log message, to allow the remote device operator to clear the punch device, or redirect the data to a specific file. This is very important for binary output since no separator cards are provided to identify the source, beginning, or termination of the data.

SENDING A QUIT SIGNAL TO THE DRIVER

Many remote terminals do not have "quit" buttons or special commands (for example, "CL" for G115/RCI protocol). Therefore, to stop the driver from printing, the remote station operator must press the STOP button (or equivalent) on the remote terminal. This disrupts the normal communications protocol and causes a quit to be signalled to the driver. This may cause one problem when using a 2780 bisync protocol; since the operator may have stopped the driver while it was printing, it cannot ask questions or print information for the operator.

Sending a quit may cause loss of locally buffered input or output. The reader should consult the manufacturer's documentation about the device. The operator can still input commands. If no commands are typed within the auto_start_delay wait time, the driver process issues an internal start command. The following commands are useful after a quit signal:

cancel	terminate and discard the current request
defer	send current request back to its queue
kill	terminate the current request
logout	log out the driver (and get ready to run a new remote station)
reinit	reinitialize the driver
release	return to normal command level (this may repeat the current request and may abort any current card input)
restart	begin the current request over again (printers go to request command level, punches restart at the current copy)
save	save the current request for possible restarting
start	resume whatever the driver process was doing at the time of the quit

If no commands are entered within 60 seconds after the driver receives a quit signal, the driver will automatically execute a start command. For some remote stations, 60 seconds is too little time; the auto_start_delay command below may be used to increase the delay time. The hold command aborts an automatic start.

DRIVER COMMAND DESCRIPTIONS

The following are commands used to invoke and manipulate I/O daemon driver processes. The conventions shown in the usage lines of these commands are the same as those used throughout the Multics manuals. For a description of these conventions, refer to Section III of the MPM Commands. For each of the driver commands that contain an underscore, the command also can be given by omitting the underscore (e.g., clean_pool or cleanpool). This allows terminals that cannot transmit an underscore to act as slaves in most cases.

As mentioned earlier in this section, these commands fall into two categories, standard driver commands and device specific driver commands. In the following paragraphs, they are listed alphabetically and are described in detail. (For a brief summary and listing by category and specific device, see Appendix B.)

auto_start_delay

auto_start_delay

Name: auto_start_delay

The auto_start_delay command displays or sets the length of time the driver will wait to issue the start command automatically after receiving a quit signal. An automatic start is cancelled if command input is received.

Usage

    auto_start_delay {N}

where N is the desired delay time in seconds. N must be at least 30 seconds. The default delay time is 60 seconds. When no argument is given, the current delay time is displayed.

banner_bars

banner_bars

Name: banner_bars

The banner_bars device specific driver command is used by printer drivers to establish how the separator bars at the bottom of the head sheet are to be printed. Printers that can overstrike should use "double" (this is the default). Other printers should use single.

Usage

    banner_bars {minor_device} {arg}

where minor_device is a minor device name (as shown by the status command) and is required if there is more than one printer minor device; arg can be one of the following separator types:

double
: overstrikes each separator line

single
: single strikes each separator line

none
: suppresses separator lines

-print
: if arg is not given, or if a single arg "-print", is given, the current value is printed

Name: banner_type

The banner_type device specific driver command is used by printer drivers to change the information printed on the front and back of each copy of a request.

Usage

banner_type {minor_device} {key}

where minor_device is a minor device name (as shown by the status command) and is required if there is more than one printer minor device; key must be one of the following:

standard
: prints the normal head and tail sheets.

none
: prints nothing except the separator bars, if required (according to the banner_bars command).

brief
: prints a short version of the head and tail sheets.

-print
: if arg is not given, or if a single arg "-print", is given, the current value is printed.

Name: cancel

The cancel command terminates the request that the driver is currently processing. The request is not placed in the coordinator's saved list and thus cannot be restarted later. This command is only valid after a quit signal, or at request command level.

After completing the command, the driver looks for another request to process. (In step mode, it returns to command level.)

Usage

cancel

Name: clean_pool

The clean_pool device specific driver command applies to drivers that can read user card decks. It allows the operator to delete all segments in the system card pool that have been there more than a specified number of days. This command is available for the master terminal only.

Usage

```
clean_pool N
```

where N is the maximum length of time in days for segments to be retained in the system card pool. All segments that have been in the card pool more than that number of days are deleted. N must be a decimal number greater than zero.

copy

copy

Name: copy

The copy device specific driver command allows the operator to set the copy number of the next copy of the current request to the value specified. This command is used only at request command level.

Usage

```
copy N
```

where N is a decimal integer between 1 and the number of copies requested by the user.

Name: ctl_term

The ctl_term command applies only to a control terminal (if attached). It allows the operator to specify the format of printed output.

One of the primary functions of the control terminal is to print information about each request processed, to aid in separating the output, and to ensure proper accountability of output generated by the driver. It is possible for the site to use preprinted forms for this purpose. (see "Using Preprinted Accountability Forms on the Control Terminal" earlier in this section.) In this case, alignment of the data on the form is very important. Generally a terminal that supports vertical tab and form feed control characters is used to ensure alignment. However, this command allows the operator to request that the software simulate the action of form feed control characters if the terminal does not provide this hardware support.

Usage

```
ctl_term arg
```

where arg falls into one of two classes: general control or simulation control (some arguments require an additional value to define the requested action):

general control

form_type STR
: specifies the format program to be used to format the data printed on the control terminal. If STR is "default", the form_type is set to the default format.

detach
: discontinues the use of the control terminal. This argument is restricted to the master terminal and is not reversible unless the reinit command is given.

simulation control

simulate
: sets the driver to simulate form feeds by software. (This argument is not reversible even by the reinit command.)

page_length N
: sets the number of lines per logical page to N. This controls the forward spacing needed to go to the top of the form.

aligned
: indicates that the forms are aligned for the purpose of form feed control. (A sample form can be printed by the sample_form command.)

Name: defer

The defer command sends the current request back to its queue marked as deferred. It is only issued from quit command level or request command level.

Usage

defer

Notes:

Requests are automatically deferred when the requested line length of the device exceeds the physical line length, or when the estimated processing time of a request exceeds the operator-defined limit (see the defer_time command below).

A deferred request will be reprocessed when the driver is given the restart_q command or when the coordinator is next initialized.

defer_time defer_time

Name: defer_time

The defer_time command is used to set or display the current time limit for automatically deferring requests.

Usage

defer_time {minor_device} {N}

where:

1. minor_device
   is the name of the minor device for which the time should be set or displayed. It is optional for drivers that have only one minor device (e.g., the central site printer). If specified, this argument must be the first argument.

2. N
   sets a new defer time in minutes, with a precision of tenths (e.g., 1.5 is one minute, 30 seconds). A time of zero indicates that infinite time is allowed. If N is not given, the current defer time and driver output rate are displayed.

Name: go

The go command makes the driver look for requests to process. If no requests are currently available, the driver asks the coordinator for a request for each "ready" device. These requests are processed as soon as they are provided by the coordinator. (This command may not be used at request command level or immediately following a quit signal.)

Usage

go {N}

where N is the number of requests processed before the driver returns to command level. If N is not specified, the driver will continue to process requests and will not return to command level until requested by the operator.

Name: halt

The halt command provides the reverse function of the ready command. It places the device or each of the specified minor devices in the inactive state. The driver does not ask the coordinator for any further requests for a halted device. However, the coordinator may have already supplied a "pending request" for the halted device. In this case, any pending request is processed immediately after the device has been halted (except when the command has been issued following a quit signal).

Usage

halt dev1 ... dev*n* {-control_arg}

where:

1. dev*i*
   is the name of a device, or minor device in the case of a multifunction device, that is to be placed in the inactive state. The device names that can be used are those printed out by the status command.

2. control_arg
   can be -all or -a to halt all devices. If the -all control argument is used, no device names need be given. No control argument is required if there is only one device for the driver. If there are multiple minor devices, the operator must specify the ones to be made inactive or else must specify -all to halt all minor devices.

Name: help

The help command prints the name of each command that may be executed by the driver. A short description of the arguments is provided with each command name. At request command level, the list of commands is limited to those unique to that command level.

Usage

help

Name: hold

The hold command is used to hold the driver at command level.

Usage

hold

Notes

When the hold command is issued from the master terminal, the slave terminal is unable to issue any command that would cause the driver to leave command level until the master terminal has issued a go command (or a start command following a quit signal). This command should always be used following a quit signal if the automatic start is to be canceled.

Name: inactive_limit

The inactive_limit command allows the I/O Daemon to log out automatically after a specified period of inactivity.

Usage

inactive_limit {N}

where N is the number of minutes of inactivity allowed. N may be from zero to 200 minutes. Zero indicates no automatic logout; this is the default. The current inactivity limit is displayed if N is not given.

Notes

The inactivity time counter is reset when a request or command is received or a quit is signalled, as well as when the driver processes a new request. A driver sitting at command level is considered active.

An inactivity logout will reinitialize a remote driver so that another station can log in and use the line.

kill kill

Name: kill

The kill command terminates the request that the driver is currently processing. The request is passed back to the coordinator and placed in the saved list where it may be restarted if desired (within the limits of the coordinator save time).

After completing the command, the driver looks for another request to process. (In step mode, it returns to command level.)

Usage

kill

Name: logout

The logout command terminates the driver process (like the standard Multics logout command).

Usage

logout

Note

When the logout command is given from a remote station, the remote driver reinitializes and gets ready to accept a new station.

Name: master

The master command is the reverse of the slave command. It allows the operator of the slave terminal to communicate with the operator of the master terminal by sending a message.

Usage

master message

where message is any arbitrary one-line message containing no more than 120 characters.

Name: new_device

The new_device command terminates the current device. The driver then asks the operator to enter a new "command or device/request_type" as described under "Driver Initialization with a Control Terminal" above.

The coordinator is notified of the termination of the current device and the device is detached by the process. If a control terminal has been attached, it also is detached.

The new_device command may only be issued from the master terminal.

Usage

```
new_device
```


Name: next

The next command specifies which request is to be taken from the queues next. This allows the operator to specify priority requests and the order in which they are to be run.

Usage

```
next -control_args
```

where the -user control argument is required and at least one other argument must be chosen from among the request identifiers (-entry, -path, and -id).

-user Person_id.Project_id
: specifies the submitter of the request by user_id. The full person and project names must be given.

-entry STR, -et STR
: specifies the entryname of the request. Starnames are not allowed. This control argument may not be used with the -path control argument.

-id ID
: specifies the match id of the request.

-path path, -pn path
: specifies the full pathname of the request. Relative pathnames and starnames are not allowed. This control argument may not be used

with the -entry control argument.

-device STR, -dev STR
: specifies which of the driver's minor devices the command is being given for. This control argument is optional for drivers with a single minor device, but is required for drivers with multiple minor devices. It serves to identify which request type the coordinator will search to find the request.

-queue N, -q N
: specifies that only queue N of the request type should be searched to find a matching request. This argument is optional; if not given, all queues will be searched.

Note

All requests to be run by the next command will be charged as though they came from queue 1.

Requests chosen to run next will be run after any restarted requests (see the restart command in this section).

This command may be given several times before a go command, to specify the exact order that requests in the queues are processed.

Name: paper_info

The paper_info device specific driver command defines the physical characteristics of the paper as used by the printer software.

Usage

```
paper_info {minor_device} {-control_args}
```

where minor_device is a minor device name (as shown by the status command) and is required if there is more than one printer minor device; control_args may be one or more of the following:

-print
: print the current values. If this is given, it must be the only control arg.

-ll N
: sets the line length to N, where N is a decimal integer from 10 to 200.

-pl N
: * sets the page length to N, where N is a decimal integer from 10 to 127.

-lpi N
: sets the number of lines per inch to N, where N is either 6 or 8.

If no control arguments are given, the current values are printed.

Note

If the printer uses a firmware VFC image, a new image is loaded (which causes the printer to go into an unsynchronized state). Otherwise, the operator is told to mount a new VFU tape.

*

Name: pause_time

The pause_time device-specific driver command allows a remote device driver to accept commands between requests by pausing a few seconds to allow the line to turn around.

Usage

```
pause_time {N}
```

where N is the number of seconds that the driver must pause between requests. N must be between 0 and 30 seconds. If N is not given, a value of 10 is assumed.

Name: print

The print device-specific driver command starts the actual printing of a file when the driver is at request command level. This command is used by printer drivers only.

Usage

```
print {N}
```

where N is a decimal integer that identifies the page at which the driver starts printing. If this argument is omitted, printing starts at the current page of the file.

Notes

A "+" or "-" preceding the page number indicates that the number is relative to the current page.

If the starting page number is beyond the end of the file, an error message is printed, and a new command is requested.

The print command causes a normal head sheet to be printed complete with separator bars if needed. The head sheet is followed by the current page of the file.

prt_control

Name: prt_control

The prt_control device specific driver command sets the driver request processing modes. Each key may be preceded by the circumflex character (^) to set the value to off.

Usage

```
prt_control {minor_device} {args}
```

where minor_device is a minor device name (as shown by the status command) and is required if there is more than one printer minor device; args may be one or more of the following:

-print
: if arg is not given, or if a single arg "-print" is given, the current modes are printed.

auto_print, ^auto_print
: This mode causes the driver to start printing each request as soon as it is received from the coordinator (after a go command has been given). This is the normal mode of operation. When ^auto_print is turned off, the driver goes to request command level immediately after printing the log message. This allows the operator to align the paper, change the paper, print sample pages, and issue all other commands allowed at request command level (including the kill command).

force_esc, ^force_esc
: This mode turns on the esc mode of the printer DIM during the processing of each request. This mode must be on if the slew-to-channel functions are to operate. Normally, the force_esc mode is set by data in the request type info (rqti) segment.

force_nep, ^force_nep
: This mode sets the noendpage (nep) mode of the printer DIM during the processing of each request, whether the user has requested that mode or not. It is normally set from data in the rqti segment. This mode is used for request types that require preprinted or preformatted paper (e.g., gummed labels, invoice forms).

force_ctl_char
: This sets the ctl_char mode of the printer DIM during the processing of each request, which allows an I/O daemon to send control sequences directly to a remote printer instead of discarding the characters or printing their octal equivalents. Setting this mode enables users who prepare print files through Compose to activate special printer features such as superscripting or multiple fonts. This mode is honored only by the remote printer driver module, remote_driver_.

If no arguments are given, the current modes are printed.

punch punch

Name: punch

The punch command is used by remote punch drivers at request command level to proceed with the punching of the requested segment.

Usage

```
punch
```

Name: pun_control

The pun_control command is used by remote drivers at normal command level to set the punch control modes. This command does not apply to the central site punch driver.

Usage

```
pun_control {minor_device} [<control_mode>]
```

where:

1. minor_device
   is the name of the punch minor device which the command is addressing. This argument is optional if there in only one punch minor device, but is required otherwise.

2. <control_mode>
   specifies the modes to be set. The mode name may be preceded by the character "^" to reset the mode. This argument is optional. If not given, the current modes for the specified minor device is printed. The following mode is currently defined:

   autopunch
   this mode allows the driver to process punch requests continuously without operator intervention. When this mode is not set (i.e., ^autopunch) the driver will come to request command level after printing the log message and wait for the operator to give the "punch" command before continuing.

   -print
   if control_mode is not given, or if a single argument "-print" is given, the modes are printed.

Notes

The ^autopunch mode is normally used by a remote operator to allow the output to be directed to a particular device based on information in the log message. Once the proper device has been assigned, the operator must type "punch" for the driver to continue with the user's request.

Name: read_cards

The read_cards device specific driver command applies to device drivers that can read user card decks. It allows the operator to input card decks from a remote station or local device. The control card format required is the same as that described under "Reading User Card Decks" in Section 4.

Usage

```
read_cards
```

Notes

The card codes that are accepted by various card readers may vary from one card reader to another. The operator should be familiar with the card codes that should be used with the card reader at the remote station

Name: ready

The ready command places the device and the specified minor devices in the active or "ready" state. The driver only requests service from the coordinator for a ready device. This command performs the reverse function of the halt command.

Usage

```
ready dev1 ... devn {-control_arg}
```

where:

1. devi
   is the name of a device, or minor device in the case of a multifunction device, that is to be placed in the ready state.

2. -control_arg
   can be -all or -a to place all devices in the ready state. If the -all control argument is used, no device names need be given. If there is only one device, no control argument is required. In this case, the ready command is executed automatically during driver initialization. If there are multiple minor devices, the operator must specify the ones to be made ready or else must specify -all to make all minor devices ready.

Name: reinit

The reinit command reinitializes the driver. The same device(s) and request type(s) are used without requesting operator input. However, remote stations have to reissue the station command and any new default request types. Also, if a control terminal is attached to the driver, its attachment, form simulation mode, and form type are retained over the reinitialization. Each device and request type is again requested from the coordinator.

The reinit command to the driver is almost the same as the standard Multics new_proc command.

Usage

reinit

release

release

Name: release

The release command returns the driver to normal command level. This command is primarily used following a quit signal. If a request was in progress, it is started over again.

Usage

release

Name: req_status

The req_status device specific driver command gives the operator information about the current request. This command may only be used at request command level.

Usage

```
req_status {-control_arg}
```

where control_arg, for printers only, may be -long or -lg to give the operator the following information:

```
number of multisegment file components
number of characters in file
current page number
current copy number
current line count
current multisegment file component
char offset in current component
char offset from start of file
printer DIM modes
printer DIM position
```

If the control argument is omitted, only the first four items in the above list are printed. In this case, the information looks like:

```
Request 10001: >print_files>invoices>Station_A.invoices
   file components:  2,    char count:  4732865
   page no:          1006  current copy no: 2
```

There is no control_arg defined for punches. The following three items are printed:

```
current copy number
current request number
current pathname
```

In this case, the information looks like:

```
Request 20001 >punch_files>invoices>Station_A.invoices
   current copy no: 2
```

Name: restart

The restart command is used either to restart processing of the current request after a device malfunction or to reprocess requests in the coordinator's saved list.

Usage

restart {arg}

where arg may be one of the following:

1. N

    is the number of the request to be restarted. The coordinator searches its saved list for a matching request. If found, the request will be re-processed ahead of any other requests, including those from the "next" command. If the request had been saved in the middle of a copy (suspended), the request will be restarted beginning at the top of the following page; a punch request will start at the beginning of that copy.

2. -from N

    specifies that all requests in the series beginning with request N are to be restarted. This is an implicit save of all requests in the series.

When the restart command is issued directly after a quit signal, with no arguments, the driver's current request is restarted. For print requests, the current page number, minus 5, and copy number are displayed and the driver goes to request command level. For punch requests, the number of copies completed (if more than one) is displayed and the operator is asked to note how many were good.

Notes

The user is charged for the requested number of copies only, regardless of how many copies were produced by this command.

If the request number series of a restarted request is still active, the driver will be switched to another series. Each restarted request is assigned a new request number, and any subsequent restart must be based on the new request number.

Name: restart_q

The restart_q command signals the coordinator to start taking requests from the beginning of the queue again. This allows any deferred requests to be run if the operator has changed the deferring criteria (see the defer_time command in this section).

Usage

restart_q {minor_device}

where minor_device is the name of one of the minor devices being run by the driver. It identifies the request type queues to be restarted. It is optional for drivers with a single minor device.

Notes

When several drivers are running from a single device class, and several requests in the queues are still in progress, it is possible that some requests will be repeated.

runout_spacing runout_spacing

Name: runout_spacing

The runout_spacing device specific driver command sets the number of lines to advance the paper after requesting a command from a remote multifunction slave terminal.

Usage

runout_spacing N

where N is the number of lines the driver advances the paper after requesting a command from the slave. N may be from zero to 60.

Note

The runout spacing is normally set in the attach description from the iod_tables. This command allows the operator to change the spacing so that driver command requests may be seen clearly above the platen.

Name: sample

The sample device specific driver command is used by printer drivers at request command level to print a sample page of the file for paper alignment or to verify the starting position in the file. The current position of a new request is always page 1. The same page may be printed as often as needed.

Usage

sample {N}

where N is the page number that the driver prints. If N is omitted, the driver prints the current page in the file.

If N is preceded by a "+" or "-", the number is relative to the current page of the file. For example, "sample +3" skips forward three pages and prints the page; "sample -8" skips backward eight pages and prints the page. Similarly, "sample 500" skips to page number 500 and prints it.

If the page number specified is beyond the end of the file, an error message is printed similar to:

```
End-of-File record encountered. EOF at page 2000, line 10.
Unable to skip to starting page.
Enter command(request):
```

and a new command is requested.

The sample command prints a page with separator bars as an aid to the operator in indicating the sample pages so they can be discarded.

Name: sample_form

The sample_form device specific driver command is used to print a sample of the data used to record request processing on the control terminal. The primary function of this command is to verify the alignment of the forms on the control terminal. The data is formatted by the program that is called for each copy of each request. (See the ctl_term command.)

Usage

```
sample_form
```

Notes

If form feed simulation is being used, the command checks to see if alignment has been set. If not, it is set before the sample form is printed.

The sample_form command applies to all drivers that use a control terminal.

sample_hs sample_hs

Name: sample_hs

The sample_hs device specific driver command prints a sample head sheet to align the paper before starting to print or after loading more paper. This command should not be used in the middle of a request (e.g., after a quit) unless the request is restarted using the restart command. Otherwise, the page restart feature of the printer driver is placed out of synchronization.

Usage

```
sample_hs {minor_device}
```

where minor_device is a minor device name (as shown by the status command) and is required if there is more than one printer minor device.

Name: save

The save command tells the coordinator that one or a series of requests are to be retained beyond the normal holding time. The action is limited to requests in the specified request number series. The save command allows requests to be saved for possible restarting until the coordinator is logged out.

Usage

save {arg}

where arg may be one of the following:

1. N

   specifies the request number in the coordinator's saved list. The coordinator searches its list of finished requests and marks the matching request number as saved for later restarting. The request remains in the saved list until the request is restarted by the restart command or until the coordinator is next initialized.

2. -from N

   specifies that all requests in the series beginning with request N are retained in the saved list.

If no argument is given, the current request will be returned to the coordinator and saved for later restarting. For printers, the request will be processed to the bottom of the next even page and a normal tail sheet will be printed, showing a charge of zero. When the request is later restarted, printing will begin at the top of the next odd page.

Notes

Once a saved request is restarted, it is not saved any longer than the normal retention time. The coordinator never deletes the user's segment while the request is being saved.

Name: sep_cards

The sep_cards command is used by a remote punch driver at normal command level to control the punching of separator cards between each output deck. If separator cards are not punched, the operator should run the driver in step mode (see the step mode command) and remove the cards from the punch as each request is completed.

Usage

```
sep_card {minor_device} {arg}
```

where:

minor_device
: is the name of the punch minor device which is being addressed. This argument is optional if there is only one punch minor device, but is required otherwise.

arg
: may be of the following:

standard
: the standard separator cards are to be punched (default).

none
: no separator cards are to be punched.

-print
: if arg is not given, or if a single arg "-print", is given, the current value is printed.

Name: single

The single device specific driver command applies only to drivers that operate a printer. It sets the single mode of the printer DIM so that form feed and vertical tab characters are treated as newline characters for the current request. It also cancels any additional requested copies that have not been processed by the driver. The single command is used after a quit to stop runaway paper feeding caused, for example, by the printing of a non-ASCII segment.

Usage

single

slave slave

Name: slave

The slave command is the reverse of the master command. It allows the master terminal operator to communicate with the operator of the slave terminal by sending a message.

Usage

slave message

where message is any arbitrary one-line message containing no more than 120 characters.

Name: slave_term

The slave_term command controls the ability of the slave terminal to enter commands, issue quit signals, and receive log or error messages. The slave terminal must be active for the command to be effective. The commands, no_commands, quits, and no_quits keys are restricted to the master terminal.

Usage

```
slave_term key
```

where key may be selected from the following:

commands
: commands can be sent from the slave terminal to the driver. (Restricted to master terminal.)

echo
: echoes each command line typed from the slave. (Input from the exec_com used by the "x" command will not be echoed.)

errors
: error messages are routed to the slave terminal.

log
: log messages are routed to the slave terminal.

modes STRs
: sets the slave terminal modes to those specified by STRs.

no_commands
: no commands can be sent from the slave terminal to the driver. (Restricted to master terminal.)

no_echo
: suppresses echoing of the slave commands (default).

no_errors
: no error messages are routed to the slave terminal.

no_log
: no log messages are routed to the slave terminal.

no_quits
: no quit signals can be sent from the slave terminal to the driver. (Restricted to master terminal.)

quits
: quit signals can be sent from the slave terminal to the driver. (Restricted to master terminal.)

Name: start

The start command allows the driver to resume operations suspended at other than the normal command level, e.g., after a quit signal. Its function is similar to the standard Multics start command. The start command cannot be issued at normal command level (see the go command).

After a quit signal, this is the only command that allows control to be returned to the point of process interruption. The action of the hold command is reset when a start command is issued.

Usage

start

Name: station

The station command is used by a driver to identify and validate a remote station. This command is similar to the standard Multics login command.

Usage

station station_id {station_password}

where:

1. station_id
   Is the registered id of the station, as defined by the administrator.

2. station_password
   Is the registered password for the remote station.

Notes

The station's identifier and password are registered in the PNT using the card input password as the station password and are supplied by the administrator for each station location.

If the remote station includes an operator's terminal with keyboard and CRT or printer, the station password may be omitted from the station command. The system will then request the station password and either suppress printing of the password or hide it with a suitable mask. This feature is particularly useful when a remote station is actually a high-quality letter printer (e.g., a Diablo 1640), where the printer is used both as the slave console and as the actual output device.

Remote stations that have no input device do not have to give a station command. However, these stations must use a dedicated phone line and have the station identifier specified in the iod_tables as described earlier for Type II remote stations.

status

status

Name: status

The status command prints information about the current status of the driver. The information provided is:

1. The I/O daemon driver version.
2. The device name and channel.
3. The request type (per minor device if more than one).
4. Whether a request is in progress and the request number.
5. The device status: ready, halted, or not attached. (If there are minor devices, this is provided per minor device.)
6. Whether there are any pending requests and their request numbers.
7. Whether step mode is set.
8. The names of any minor devices (to be used with the ready and halt commands).

Usage

```
status {-control_arg}
```

where control_arg may be -long or -lg to print the status of inactive minor devices (devices that cannot be made ready).

Name: step

The step command either sets (puts the driver into) or resets (takes the driver out of) step mode. When in step mode, the driver returns to command level after processing each request from the coordinator. When not in step mode, the driver processes requests from the coordinator as soon as received without operator interaction. Step mode is useful for checking the alignment of paper on the printer or other device functions prior to allowing the driver to run continuously without operator interaction.

Usage

    step {arg}

where arg can be "set" or "reset" to put the driver into or take the driver out of step mode. If no argument is supplied, step mode is set. The driver is not in step mode immediately after driver initialization.

Name: x

The x command allows drivers to execute an admin exec_com on a site-defined basis.

Usage

    x function {args}

where:

1. function
   is a site-defined function name.

2. args
   are any arguments needed to implement function.

Notes

When the user issues the x command, the driver constructs the command line:

exec_com >ddd>idd>NAME function {args}

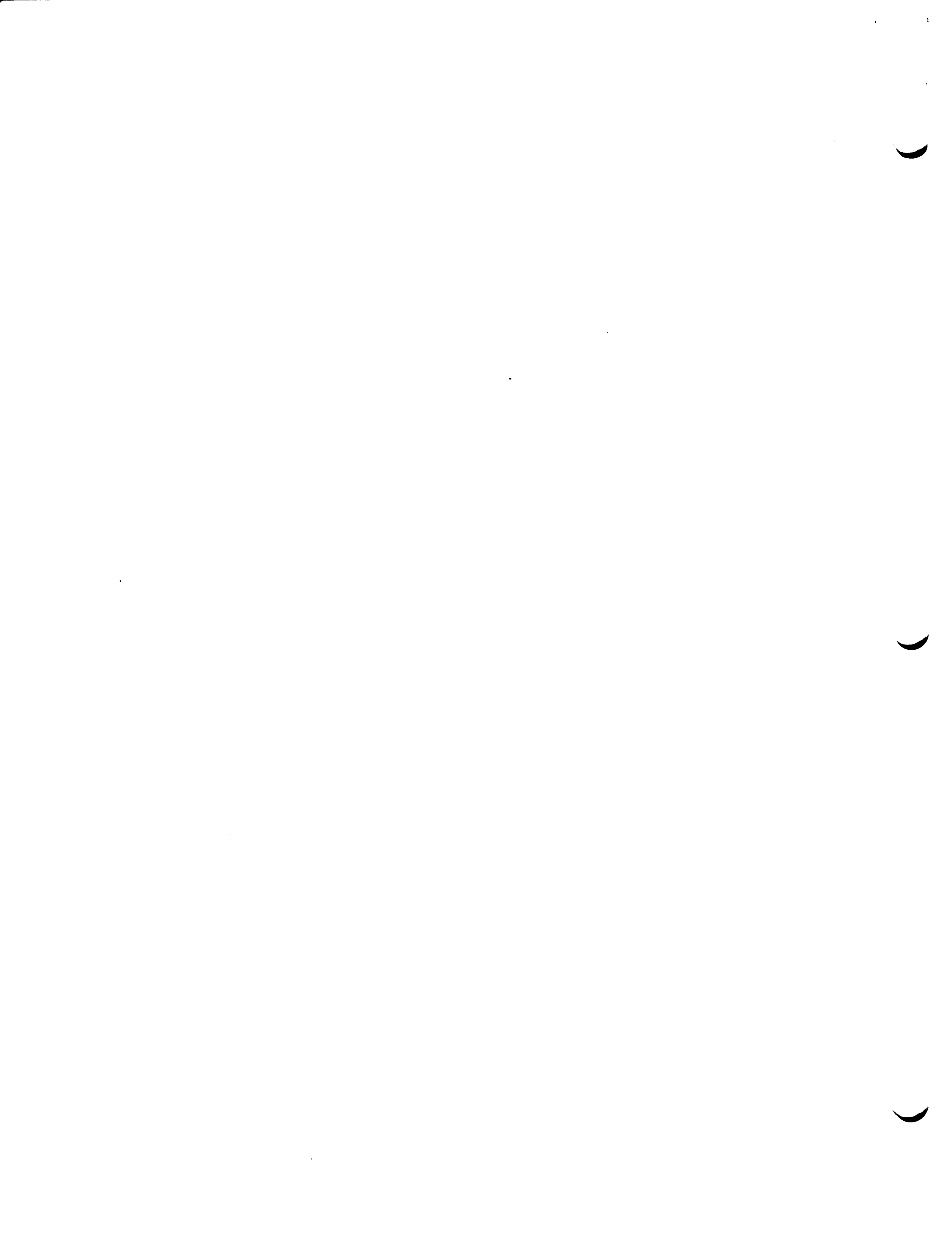
where function and args are as above; NAME is either <major_device>_admin.ec for standard drivers or <station_id>_admin.ec for remote drivers. If NAME is not found, the driver will look for the default of iod_admin.ec (see Appendix C). Added names can be used to group exec_coms into categories.

Drivers that run as IO.SysDaemon have a great deal of access to the storage system. Administrators must be careful in choosing commands for the admin exec_coms to avoid accidents or vandalism.

The Multics command iod_command may be used within an admin exec_com to execute arbitrary I/O daemon commands. For example:

iod_command defer_time 30

may be used in an admin exec_com to change the auto defer time limit for the current driver to 30 minutes. The iod_command command is described in detail in Appendix A.



SECTION 4

MANAGEMENT OF CARD INPUT STATION

CARD INPUT ACCESS CONTROL

Card input is subjected to special access control checks in order to provide for security. This section describes the checks and the tables and segments that support them.

Card Input Password

A card input password for each user, separate from that user's interactive password, is stored in the system Password Name Table (PNT). This password is assigned when a user is registered, and can be changed either by the user or a system administrator.

Users without a card input password who have r access to card_input_password.acs and the associated <station>.acs in the directory >system_control_1>rcp may also use bulk data input, if they have created a card_input.acs segment in their mailbox directory (see below). In addition, the card reading process must have rw access to >system_control_1>PNT.

Registering Card Input Users

Card input users are registered with the register command. For a description of the register command see the MAM Accounting manual, Order No. AS68.

Remote Job Entry Submission Access Control

Any process (e.g., IO.SysDaemon and Card_input.Daemon) which is to read and process remote job entry (RJE) card input must have e access to the segment:

>system_control_1>proxy>absentee_proxy.acs

to be able to submit proxy absentee requests.

Station Registration and Password

Each card input station (central or remote) is registered in the PNT with the register command. A card input password is associated with each station.

The station card input password must be specified by the operator as part of the sign-on sequence.

User Card Input Access Segment

Users must explicitly permit a station to submit card input for them by creating the segment "card_input.acs" in their mailbox directory:

>user_dir_dir>Project_id>Person_id

The ACL for this segment must give r access to each station that the user permits to submit bulk data input and e access for each station that the user permits to submit RJE jobs. For example:

re Station_A.*.*

The card reading process must have s access to the project_id and person_id directories. If this segment does not exist or if the access is not as specified, the card input is aborted.

System Station Access Control Segment

All users allowed to submit card input from a station must be on the ACL of the stations access control segment (e.g., Station_A.acs). Each card input station must have an access control segment residing in the directory >system_control_1>rcp. The card reading process must have s access to this directory. If access is not specified, or if this segment does not exist, the card input is aborted.

The star convention may be used in the normal fashion, for example:

```
r    user.*.*
re   *.Project_A.*
n    *.*.*
```

This check allows a site to specify that a certain station is reserved for the use of a certain group of users, perhaps those who pay for the equipment. It can also be used to ensure that certain stations are not used to submit RJE card input for privileged users, such as *.SysAdmin, who should never normally use the facility.

READING USER CARD DECKS

Remote terminals with card readers and the central site reader can be used to enter user card decks for both bulk data input and RJE. The operator must issue the read_cards command to the card reading process to start reading in user card decks. The card reading process may be either the central site Card_input.Daemon (see "Reading Cards at the Central Site" below) or an I/O daemon driver using the remote_driver_ device driver module (see "Standard Driver Modules" in Section 2).

Because different card readers have different punch card decoding conventions, the user is warned that the same character may require different punch codes on different readers. Thus, for example, a + character may be represented by a 12-8-2 punch on one reader, and a 12-0 on another. Obviously, cards prepared for the first reader do not transmit the same data on the second reader and may in fact be unreadable. The user should consult the manufacturer's documentation before preparing any cards for input.

This page intentionally left blank.

To prepare the user's card decks for reading, the operator must place pairs of EOF and UID control cards in front and back of each deck. The control card format, for operator supplied control cards, is as follows:

```
Column 1                                    Column 80
   ↓                                            ↓

   ++EOF

   ++UID <uid_string>

   ++END
```

The card with ++EOF (starting in column one) is tne end-of-file marker. It must be the first card placed in the card reader after the read_cards command is given.

The card with ++UID <uid_string> is a unique ID card; it must always follow an EOF card. The <uid_string> can be any string of characters (except spaces) of 1 to 12 characters long (see example below). The <uid_string> on the ++UID cards at the front and back of each user's card deck must be identical. These cards are used to separate user card decks. Therefore, the operator should keep about 10 pairs of matching ++UID cards near the card reader so that he is able to stack several user card decks in the hopper at one time.

The ++END card is used to terminate the reading of cards and allows the process to return to normal command level. If a ++END card is not placed after the last ++UID card (or after and ++EOF card) the driver assumes that there are more card decks to be read and waits for them to be loaded into the card reader.

A complete group of card decks, ready for reading, would appear as follows:

```
++EOF
++UID ZZZZZ
         .
(First User Card Deck)
         .
++EOF
++UID ZZZZZ
++EOF
++UID ABCDEF
         .
(Next User Card Deck)
         .
++EOF
++UID ABCDEF
         .
         .
         .
++EOF
++UID 1234567890
         .
(Last user Card Deck)
         .
++EOF
++UID 1234567890
++END
```

The user must submit a complete card deck to operations. The deck must follow the format specified in Appendix C of the MPM Reference Guide. For the convenience of the operator, the general format is described here briefly.

MINIMUM FORMAT OF A CARD DECK FOR BULK DATA INPUT

```
++DATA  <deck_name> <Person_id> <Project_id>
++PASSWORD  <xxxxxxxx>
++INPUT
   .
   .
(user data cards)
   .
   .
```

MINIMUM FORMAT OF A CARD DECK FOR REMOTE JOB ENTRY

```
++RJE  <deck_name> <Person_id> <Project_id>
++PASSWORD  <xxxxxxx>
++INPUT
   .
   .
(user absentee commands)
   .
   .
```

Only the first two cards and the ++INPUT card are required for each deck. All other cards are optional. Examples of optional cards that may appear in a bulk input deck are:

```
++AIM <deck access class>
++FORMAT  <punch format> <format control modes>
++CONTROL OVERWRITE
```

Examples of optional cards that may appear in an RJE input deck are:

```
++AIM   <deck access class>
++FORMAT   <punch format> <format control modes>
++RJECONTROL   <ear control args>
++RJEARGS   <ear args>
++EPILOGUE   <command line>
++ABSIN   <pathname>
```

**
**

For more information, see Appendix C of the MPM Reference Guide.

NOTE: The access class specified on the ++AIM cards of each user deck MUST match the authorization of the card reading process. Normally, the authorization is system_low and the ++AIM cards can be omitted. However, if the authorization is greater than system_low, the ++AIM cards are required and must match the authorization, or the card reading aborts. So, the operator should always check user decks for correct ++AIM cards.

READING CARDS AT THE CENTRAL SITE

In order for a Multics user to read data on punched cards into the Multics system, those cards normally must be submitted to operations. Users normally cannot use the central site card reader on their own, without operator intervention. The central site card reader is controlled by a Multics daemon named Card_Input.Daemon. This daemon is responsible for driving the card reader, transferring the data into online storage, submitting local RJE jobs, reporting errors, and communicating with the operator. The card daemon may use

the message coordinator to run as a "consoleless" daemon or may be logged in from its own terminal.

Further information may be found under "Bulk Input/Output" in Section V of the MPM Reference Guide and in the description of the card reader hardware.

Login and Initialization

The card input daemon is logged in like any other io daemon. Once logged in it accepts a subset of the standard daemon commands, described below. The authorization specified at login must be the same as the access class of the card decks which the daemon is expected to process.

Communicating with the Daemon

When the central site card daemon requires instructions from the operator, it types:

```
Card Daemon:  Command?
```

This occurs after initialization, after reading an end card, after a quit signal, or after some error condition is encountered. The following commands are understood by the daemon and may be typed on the daemon terminal:

help
: print a short description of available commands.

read_cards
: start reading cards from the card reader. The daemon assumes that the reader is ready (or waits for it after printing a message).

start
: continue the operation in progress after having received a quit signal.

logout
: logout the daemon.

reinit
: attempt to reinitialize the card daemon by detaching the card reader and reattaching it. This command may be used if the daemon appears to be in an inconsistent state.

abort
: terminate the reading of the current load of card decks after a quit signal has been received. To kill only the current card deck instead of the whole load, immediately follow the abort command with the read_cards command, and other decks in the card reader are processed correctly.

clean_pool
: delete old card deck copies stored in the system storage areas. This command causes the daemon to ask for the age of segments to be deleted. (This command is normally used to the request of the system administrator or in the event of a record quota overflow.)

Error Conditions

The card daemon attempts to recover from most errors involving incorrectly punched control cards by forward spacing to the next card deck. That is, if an error occurs during the reading of a card deck, that deck is skipped and the reading continues with the next card deck. When the card daemon encounters a deck having an access class (as specified on the ++AIM card) that is different from its own access authorization, the card daemon stops the card reader and requires that the problem be corrected before continuing.

READING CARDS AT THE REMOTE SITE

When the remote station operator has completed the station initialization described in Section 3, the remote station accepts card input (either bulk card input or RJE) after the operator enters the command:

```
read_cards
```

Log and error messages are output to the terminal as each deck is processed. Error and status messages are sent to the submitter of the card deck via the mail facility.

APPENDIX A

ADMINISTRATIVE COMMANDS AND ACTIVE FUNCTIONS

This appendix contains descriptions of commands and active functions needed by an administrator to manage the I/O daemon.

The conventions shown in the usage lines of these commands are the same as those used throughout the set of Multics manuals; briefly, arguments enclosed in braces ({}) are optional and all others are required. For a complete description of all of the usage line conventions, refer to Section III of the MPM Commands.

Name: create_daemon_queues, cdq

The create_daemon_queues command creates the I/O daemon queues. It determines which queues to create by examining the iod_tables segment.

Usage

```
create_daemon_queues {path} {-control_args}
```

where:

1. path
   is the pathname of an iod_tables segment created by the iod_tables_compiler. The queues are created in the containing directory of path, using the request types specified by the iod_tables segment. This argument is optional.

2. control_arg
   can be one of the following:

   -directory path, -dr path
   queues are created in the directory whose pathname is path. This control argument is provided for testing purposes only; normally, it should be omitted. When not specified, the queues are created in the >daemon_dir_dir>io_daemon_dir directory. This argument may not be given with a path specification.

   -reset_access
   resets the ACLs on each queue to the default value, if the queue already exists.

Notes

The I/O daemon tables segment, called iod_tables, is expected to be found in the same directory in which the queues are to be created. For each request type defined in iod_tables, one to four queues are created, depending on the maximum number of queues for that request type (as defined in iod_tables). The name of each queue is of the form XXX_N.ms where XXX is the associated request type name and N is the priority number of the queue. The ms suffix indicates that each queue is a message segment.

For further details, see the discussion of the I/O daemon tables in Section 2.

Name: cv_prt_rqti

The cv_prt_rqti command converts an ASCII printer request type info source segment into a printer request type info segment (rqti segment) for use by the I/O daemon. The newly converted rqti segment is placed in the current working directory. The entryname of the new rqti segment is the same as the entryname of its source segment without the rqti suffix.

Usage

```
cv_prt_rqti path {-control_arg}
```

where:

1. path
   is the pathname of the request type info source segment. The source segment must have a rqti suffix, although the suffix may be omitted in the command invocation.

2. control_arg
   can be one of the following:

   -brief, -bf
   prints error messages in the short format.

   -long, -lg
   prints error messages in the long format. This is the default.

Notes

For a description of the syntax of a request type info source segment and an example segment refer to Section 2.

Example

The command line:

```
cv_prt_rqti printer_info.rqti
```

creates a request type info segment named printer_info in the working directory.

Name: display_prt_rqti

The display_prt_rqti command interprets the contents of an I/O daemon printer request type info segment (rqti segment) and displays all defined values. The output format is such that when directed to a file, the file may be used as input to the cv_prt_rqti command.

Usage

```
display_prt_rqti path
```

where path is the pathname of the printer request type info segment.

Name: iod_command

The iod_command command permits execution of I/O daemon commands from within admin exec_coms invoked by the I/O daemon x command.

Usage

```
iod_command io_daemon_command {args}
```

where:

1. io_daemon_command
   is the I/O daemon command to be executed.

2. args
   are any arguments needed to implement the specified command.

Note

The go command may not be issued using iod_command.

Example

```
iod_command defer_time pica_10 30
```

may be used within an I/O daemon admin exec_com to set the auto defer time of the pica_10 minor device of the current I/O daemon driver to 30 minutes.

This page intentionally left blank.

Name: iod_tables_compiler

The iod_tables_compiler command is the translator for the I/O daemon tables source language (described in Section 2). Source segments to be translated by iod_tables_compiler must have a name ending with the suffix iodt. The name of an object segment produced by iod_tables_compiler is the same as that of the corresponding source segment with the iodt suffix removed. The object segment is placed in the working directory.

Usage

```
iod_tables_compiler path
```

where path is the relative or absolute pathname of the source segment to be translated.

Name: iod_val

The iod_val active function supplies several preset driver parameters to be used in driver admin exec_coms. Site administrators use the iod_val active function in conjunction with the driver x command to set up and modify these exec_coms.

Usage

[iod_val key]

where key is a character string parameter name associated with the value to be returned. The key, defined during initialization of the given driver, may be one of the following:

For all standard drivers:

device
: the name of the major device that the driver is running.

station_id
: the name of the station_id that the driver is running (equivalent to the major device). The default is the name of the major device if the station is not a remote device.

request_type
: the name of the request type that is being run on the driver.

channel
: the name of the iom or tty channel of the driver.

<minor device>
: the name of the request type that is being processed on the minor device.

rqt_string
: a string of request type names, separated by spaces, of all (printer, punch, etc) request types the driver can process. This key is equivalent to the request_type key if the driver is running only one minor device.

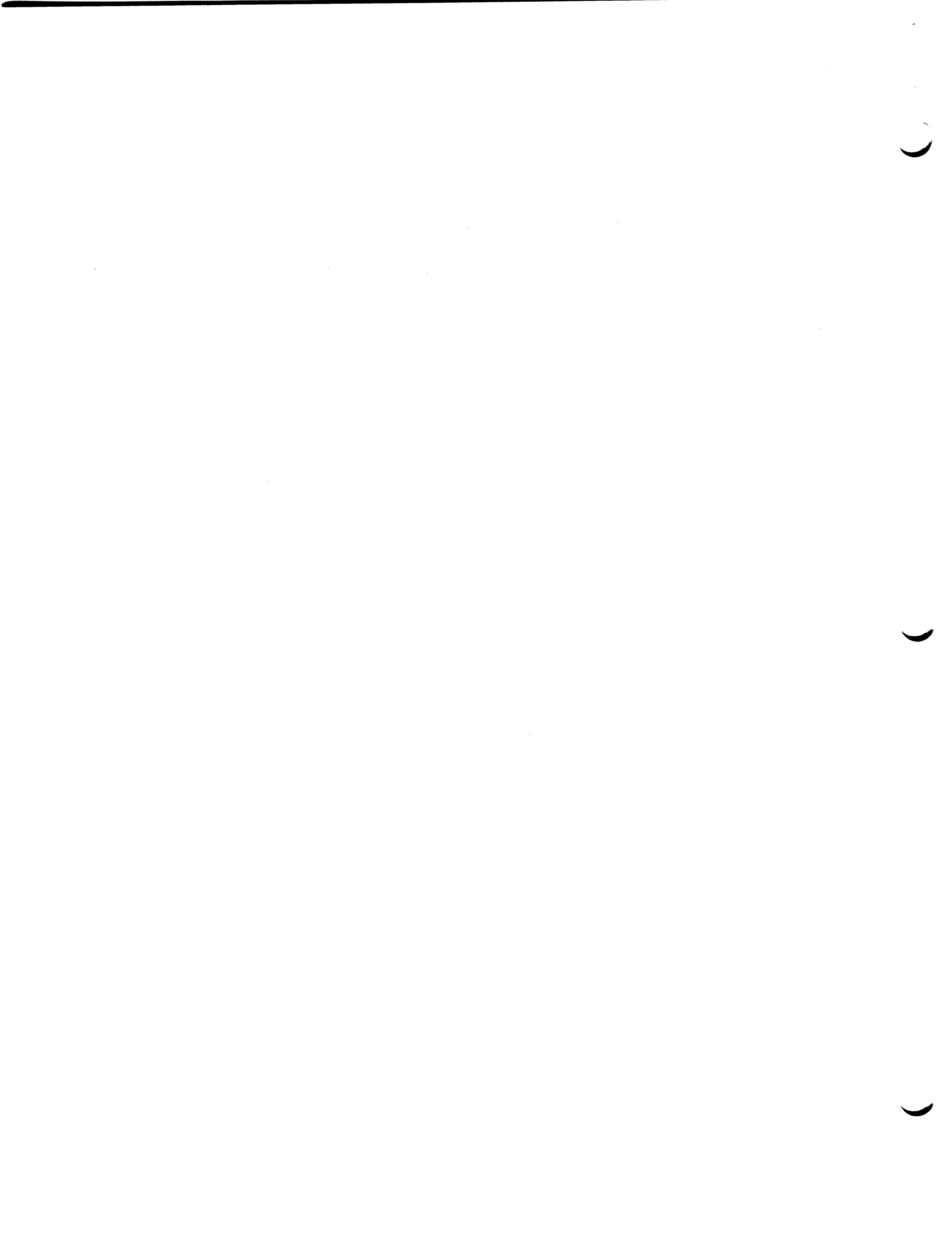
For remote drivers:

request_type
: the request type for a single printer device, if present.

pun_rqt
: the request type for a single punch device, if present.

Notes

If a key is given that has not been defined, the string "undefined!" is returned.



Name: print_devices

The print_devices command prints a list of devices for each request type handled by the I/O daemon. Also, the driver access name and driver authorization (if any) for each request type are printed. An asterisk (*) immediately preceding a device name indicates that the associated request type is the default for the device.

Usage

```
print_devices {-control_args}
```

where control_args can be one or more of the following:

-brief, -bf
: suppresses printing of a heading line.

-access_name STR, -an STR
: lists only devices for those request types having a driver access name of STR (STR should be of the form Person_id.Project_id).

-request_type STR, -rqt STR
: lists only devices for the request type specified by STR (e.g., printer, punch).

-dir path
: specifies the absolute pathname of the directory containing the iod_working_tables segment. If not given, the directory >ddd>idd is assumed.

Name: print_iod_tables

The print_iod_tables command displays the contents of an object segment produced by the iod_tables_compiler command. The format of the output corresponds exactly to the source language accepted by the iod_tables_compiler command. In fact, if the output of the print_iod_tables command is directed to a segment, the resulting segment can be translated by the iod_tables_compiler command.

Usage

```
print_iod_tables path
```

where path is the relative or absolute pathname of the object segment to be displayed.

Name: print_line_ids

The print_line_ids command prints a list of logical line_ids and their associated communications channel from data in the iod_working_tables segment.

Usage

print_line_ids {-control_args}

where:

1. -brief, -bf
   suppresses printing of a heading line.

2. -dir path
   specifies the absolute pathname of the directory containing the iod_working_tables segment. If not given, the directory >ddd>idd is assumed.

Name: print_spooling_tape

The print_spooling_tape command directly attaches a printer and prints the contents of a tape written by the spool driver.

Usage

```
print_spooling_tape prtdim device {-control_args}
```

where:

1. prtdim
   is the literal string "prtdim", which is the name of the standard Multics printer I/O module (DIM).

2. device
   is the name of the IOM channel for the printer device to use.

3. control_args
   are chosen from the following:

   -number N, -nbr N
   begins printing at N where N is the file number of a file on tape. If it is omitted, printing begins with the first file on the spooling tape.

   -debug, -db
   turns on audit trace during printing. The default is debug off.

Example

To print a spooling tape, starting with the third file on the tape, using the standard Multics printer I/O module (identified as "prtdim") and the printer (identified as "prta"), the operator types:

```
print_spooling_tape   prtdim   prta  -nbr 3
```

Then the operator is asked for volume identifiers and spooling limits, as shown below:

```
Enter volids and optional file limits:
```

The operator types:

```
-volid SPOOL1 -files 50
```

giving the volid of the spooling tape to be printed (SPOOL1) and a limit of 50 files to print before printing is stopped. The I/O module, tape_ansi_, determines whether to read the spooling volume at 800 or 1600 bpi density and, from the tape labels, the I/O module determines the tape block size and maximum line-length to be printed.

Next, the operator is requested to mount the first volid and a message is typed on the terminal as follows:

```
Mounting volume SPOOL1 with no write ring.
Volume mounted on tape_XX.
```

As each file on the spooling tape is printed, a message appears on the terminal giving the number of the file. This continues until the file limit has been reached or until the entire tape has been processed. The spool driver output looks like the following:

```
Printing FILE 3
Printing FILE 4
 .
 .
 .
Printing FILE 50

Reached end of data for current fileset.

Taking current volume down.

Printer detached.

Processing of spooling tape ended.

Spooling file count is 48
Spooling line count is 1254
```

At this point, printing has finished and the operator can logout the process.

Description of the Spooling Tape

The spool driver creates either an 800 or 1600 bpi ANSI standard tape (ASCII) with D-format (variable length) records of a specified printer line length, that are blocked to 8192 characters, unless the interchange option is specified, in which case, the block size is 2048 characters and the density is 800 bpi. Each print request constitutes one ANSI tape file, which is surrounded by ANSI standard tape labels. The exact format of the ANSI tape can be found by referring to Draft Proposed Revision X3L5/419T of the American National Standard Institute's ANSI X3.27-1969, "Magnetic Tape Labels and File Structure for Information Interchange." Each line (logical record) of the request (print file) is preceded by a USA printer carriage control character that directs a printer action before the line is printed. These control characters and the corresponding Multics spool driver slew functions are listed below.

USA Char	Spool Driver Slew Function	Printer Action
blank	NL	One line spaced
0	2(NL)	Two lines spaced
-	3(NL)	Three lines spaced
+	CR	Suppress line space
1	FF	Skip to channel 1 (top: line 3, any page)
2	none	Skip to channel 2
3	none	Skip to channel 3
4	none	Skip to channel 4
5	none	Skip to channel 5
6	none	Skip to channel 6
7	bottom inside page	Skip to channel 7 (odd page)
8	bottom inside page	Skip to channel 8 (even page)
9	none	Skip to channel 9
A	none	Skip to channel 10
B	none	Skip to channel 11
C	none	Skip to channel 12

Note: The printer action occurs before a line occurs printed.

APPENDIX B

SUMMARY OF I/O DAEMON COMMANDS

The following is a list of the commands used to control I/O daemon driver processes. Standard driver commands and device specific driver commands are described; the latter are broken down by device types. In the list, the name of each command is followed by its usage line and a brief description of the function of the command. All of the commands described below may also be issued without the embedded underscores, e.g., cleanpool or clean_pool.

For more information, consult the specific driver documentation in Section 3 or use the help command.

STANDARD DRIVER COMMANDS

auto_start_delay
  Usage: auto_start_delay {N}
  sets wait time between quit signal and automatic start command

cancel
  Usage: cancel
  terminates the request the driver is currently processing (nonrestartable)

ctl_term
  Usage: ctl_term arg
  allows the operator to specify actions of a control terminal

defer
  Usage: defer
  sends current request back to its queue

defer_time
  Usage: defer_time {minor_device} {N}
  sets time limit for automatically deferring requests

go
  Usage: go {N}
  causes the driver to look for requests to process

halt
  Usage: halt dev1 ... devn {-control_arg}
  places the device(s) in the inactive state

help
  Usage: help
  prints the name of each command that may be executed by the driver

hold
  Usage: hold
  holds the driver at command level

inactive_limit
Usage: inactive_limit {N}
sets time limit for inactivity logout

kill
Usage: kill
terminates the request the driver is currently processing (restartable)

logout
Usage: logout
causes the driver process to log out

master
Usage: master message
allows the slave terminal operator to communicate with the operator of the master terminal

new_device
Usage: new_device
allows the operator to terminate the current device

next
Usage: next -control_args
processes a specified request next

ready
Usage: ready dev1 ... devn {-control_arg}
places the device(s) in the active state

reinit
Usage: reinit
reinitializes the driver

release
Usage: release
returns the driver to normal command level from any other command level

restart
Usage: restart arg
restarts processing of the current request or specified requests

restart_q
Usage: restart_q {minor_device}
tells coordinator to reexamine any deferred requests

save
Usage: save arg
suspends current request or keeps specified requests in the saved list beyond the normal holding time

slave
Usage: slave message
allows the master terminal operator to communicate with the operator of the slave terminal

slave_term
Usage: slave_term key
controls the actions of the slave terminal

start
Usage: start
allows the driver to continue operation at the point where it was interrupted by the quit signal

station
Usage: station station_id password
identifies a remote station at login

status
    Usage: status {-control_arg}
    prints information about the current status of the driver

step
    Usage: step {arg}
    puts the driver into, or takes it out of, step mode

x
    Usage: x function {args}
    executes site-defined functions from admin exec_com

DEVICE SPECIFIC DRIVER COMMANDS

Commands for Printers

banner_bars
    Usage: banner_bars {minor_device} {arg}
    allows the operator to specify how the separator bars are to be printed

banner_type
    Usage: banner_type {minor_device} {key}
    allows the operator to specify what will be printed on head and tail banner sheets

paper_info
    Usage: paper_info {minor_device} {-control_args}
    allows the operator to specify line length, page length, and lines per inch to be used in printing

prt_control
    Usage: prt_control {minor_device} {args}
    allows the operator to set the driver request processing modes

sample_hs
    Usage: sample_hs {minor_device}
    allows the operator to print a sample head sheet to align the paper

single
    Usage: single
    allows the operator to single space on formfeed and vertical tab

Commands for Printers at Request Command Level Only

copy
    Usage: copy N
    sets the copy number of the next copy to be printed to N

print
    Usage: print {N}
    starts printing the next copy from the current page or page N

req_status
    Usage: req_status {-control_arg}
    prints status information about the current request

sample
    Usage: sample {N}
    prints a sample of the current page or page N

Commands for Local Punches

No special punch device commands are required; standard commands may be used.

Commands for Remote Punches

pun_control
    Usage: pun_control {minor_device} {<control_mode>}
    sets the punch control modes (does not apply to the central site punch driver).

sep_cards
    Usage: {minor_device} {arg}
    controls punching of separator cards between each output deck

Command for Remote Punches at Request Command Level Only

copy
    Usage: copy N
    sets the copy number of the next copy to be printed to N

punch
    Usage: punch
    proceeds with the punching of the requested segment

req_status
    Usage: req_status
    prints status information about the current request

Commands for Card Input

clean_pool
    Usage: clean_pool N
    allows the operator to delete all segments in the system card pool that have been there more than a specified number of days

read_cards
    Usage: read_cards
    allows the operator to input card decks from a remote station or local device

Commands for Control Terminal Operation (Most Drivers)

sample_form
    Usage: sample_form
    prints on the control terminal a sample of the data used to record request processing

Commands for Remote Device Control

pause_time
Usage: pause_time N
sets pause time, in seconds, between requests in order to accept input

runout_spacing
Usage: runout_spacing N
sets paper advance after requesting a command from a remote multifunction slave terminal

Commands for Spool Driver

Spool driver commands are a subset of those for the printers described under "Commands for Printers" above. The spool driver commands are:

banner_bars
paper_info
prt_control
sample_hs
single

This page intentionally left blank.

APPENDIX C

I/O DAEMON ADMIN EXEC_COM FORMAT

An I/O daemon admin exec_com is written by a site administrator to provide site-defined driver x command functions. The use of admin exec_coms is optional, but when missing, the driver x command will not work. See Appendix E for the application of the admin exec_com to the creation of a driver-to-driver message facility.

Each I/O daemon admin exec_com is located in the >ddd>idd directory and follows standard exec_com rules. There are two types of admin exec_coms: general and device specific. These differ only in segment name, to allow the site to separate x command functions by device name (station_id for remote stations). The iod_admin.ec segment is the general exec_com and will be used by any driver that cannot find a device-specific exec_com. A <device>_admin.ec segment is a device-specific exec_com for the given major device; for example, prta_admin.ec is specific to device prta. Added names can be used to group several devices under a single device-specific exec_com.

The Multics command iod_command may be used within an admin exec_com to execute arbitrary I/O daemon commands. For example:

```
iod_command defer_time 30
```

may be used in an admin exec_com to change the auto defer time limit for the current driver to 30 minutes. The iod_command command is described in detail in Appendix A.

When writing an I/O daemon admin exec_com, the administrator must remember that the process that executes it will, most likely, have full SysDaemon access and privileges to the system. Therefore, care must be given in choosing what functions should be placed at the hands of a remote station operator or an inexperienced device operator.

The remainder of this appendix is a sample section of an admin exec_com. It includes examples of how some iod_val active function keys can be used to protect against operator errors. This sample is for illustration only; see the iod_admin.ec segment supplied in the release for working purposes.

```
& ----------------------------------------------------------
&
& iod_admin.ec   (to be found in >ddd>idd)
&
& This is the exec_com for the IO Daemon driver "x" command.
& The first argument to the "x" command is &1 in this exec_com.
& The standard action is to transfer control to a label
& which will implement the function of &1.
&
& Any arguments associated with an "x" command function begin
& with &2 in this exec_com.

&command_line off
&goto &1.command

&label help.command
&
& For "x help" print a list of x command functions.
&
&print cdr -user Pers.Proj <seg_ident>
&print car -user Pers.Proj <seg_ident>
&print pq {ldr_args}
&quit

&label cdr.command
&
& For "x cdr -user Pers.Proj <seg_ident>"
$ to cancel a dprint request for this driver
&
&if [not [exists argument &2]]
&then &goto missing_arg.error
cdr -rqt [iod_val request_type] &f2
&quit

&label car.command
&
& For "x car -user Pers.Proj <seg_ident>"
& to cancel an RJE job sent by this station
&
&if [not [exists argument &2]]
&then go to missing_arg.error
car -sender [iod_val station] &f2
&quit

&label pq.command
&
& For "x pq {ldr_args}"
& to list all requests that can be processed by this driver
&
&if [exists argument &2]
&then ldr -a &f2
&else ldr -a -admin -rqt ([iod_val rqt_string]) -tt
&quit

&label &1.command
&
& This is a catchall for any undefined command functions.
&
&print Undefined driver x command function.
&
ioa_ "received command: ^(^a ^)" &f1
&
&quit

&label missing_arg.error
&
&print Expected argument missing. Try again or type "x help".
&
&quit
```

APPENDIX D

GENERATING A DRIVER PROCESS IN TEST MODE

This appendix describes how to generate a driver process in a test environment. A working knowledge of the system software (commands, bound object segments, archives, etc.) is assumed. The information provided here is to be used only as a guide and is not intended to cover all circumstances and requirements.

The test environment allows a user to test out changes to software and data bases (ttt, rqti segments, iod_tables, etc.) normally used by the system coordinator and drivers, both remote and on-site. The test environment includes more detail in error messages, and special commands which control the test process. Full use of the Multics command language is provided, enabling the user to set break points using either the probe or the debug command.

TEST DIRECTORY STRUCTURE

The test directory structure is similar to that of the >ddd>idd directory. Throughout this appendix, the directory's pathname is indicated by the term TEST_DIR. The test directory can be located anywhere that the user has sma access. Also, some system data bases can be shared with those of the system daemons.

If any request type is configured to use a request type info segment (rqti), the rqt_info_segs directory must be created by the user in the test directory. This directory must contain all rqti segments to be used in the test session.

If card input is to be performed, the card_pool directory must be created by the user in the test directory. This directory has a different name than the cards directory used by a standard driver and has a different relative location in the drivers' directory structure. This directory must have sufficient quota assigned to it to handle whatever card input is to be performed. Directories and quota are managed in the same manner as for the >daemon_dir_dir>cards directory.

The segments and directories created by the coordinator in the test directory are identical to those normally created in the >daemon_dir_dir>io_daemon_dir directory. Refer to Section 2 for more information about this directory.

User Generated Data Bases

In TEST_DIR the user creates the segment iod_tables.iodt (described in Section 2). This segment must be compiled by the iod_tables_compiler command (described in Appendix A) to create the iod_tables segment.

If the driver is run from other than an IO.SysDaemon process, the following must be present in the iod_tables.iodt segment for each request type used:

```
driver_userid:      Person_id.Project_id;
accounting:         nothing;
```

where Person_id.Project_id identifies the testing process. The nothing command (see the Multics System Programming Tools manual, Order No. AZ03) will be called instead of the charge_user_ subroutine so that actual charges will be ignored. If testing an accounting routine, its name should be given.

The user is required to create message segment queues for the request types that will be used in the test session. This can be done by using the create_daemon_queues command (described in Appendix A) or manually by using the message segment commands (described in the MAM System). When using the create_daemon_queues command, the -dr path control argument must be given:

```
create_daemon_queues -dr TEST_DIR
```

If testing remote devices, the user's process must have the dialok attribute in the PDT and correct access to the access control segment for the communications channel or peripheral device; see the system administrator for assistance.

The user may optionally use a different terminal type table (TTT) than the system TTT. Refer to the MPM Communications Input/Output manual, Order No. CC92, for a description of how to set up a TTT and the associated commands.

If the x command is used, an iod admin exec_com or device admin exec_com must be included in TEST_DIR (see Appendix!C).

Shared Data Bases

The test process can share some data bases with the standard system daemon drivers.

The file "PNT" in the >system_control_1 directory is used by the test process to check station identifiers, passwords, and card input users; alternatively, a copy of PNT may be used by issuing the command:

```
validate_card_input_$test TEST_DIR
```

This data base is normally maintained by the system administrator.

The required access control segments for card input are also the same ones used by the system drivers. The testing process must have the same access to these segments as a regular driver process.

MANIPULATING REQUESTS IN THE TEST QUEUES

Since the test driver process will be using message segments in the test directory, the dprint, dpunch, list_daemon_requests (ldr) and cancel_daemon_requests (cdr) commands must be made aware of the test environment. This is done by calling special entries in each command procedure and indicating the test directory as follows:

```
dprint_$test TEST_DIR
ldr$test_ldr TEST_DIR
cdr$test_cdr TEST_DIR
```

This page intentionally left blank.

Once this is done, the normal system printer/punch queues are no longer known to the test process. Issuing the new_proc command is one method of restoring access to the normal system queues; the user could also issue the above commands with the pathname >ddd>idd.

THE TEST PROCESS

A standard I/O daemon process operates either as a coordinator or as a driver, and a check is made so that only one coordinator is operating on the system at one time. In test mode, a single test process may perform the functions of both coordinator and driver; or, after one interactive test process has become a coordinator, another interactive process may become a driver. The second interactive process must use the same test directory as the first process. The test processes acting as coordinator and driver, are unknown to the standard system I/O daemon processes.

Experimental software should exist in either bound or loose form in the test directory. If one component of a bound object segment is loose, then all components must be loose. The user may want to initiate each object segment first.

The test process is started by calling the test entry of the iod_overseer_ subroutine:

```
test_io_daemon -dr TEST_DIR
```

When running the coordinator and driver in a single test process, the dialog from this point looks like the following, with user responses preceded by an exclamation point (!):

```
  Enter command:  coordinator or driver
! coord
  I/O Coordinator Version: X.X
  I/O Coordinator initialized
! driver

  I/O Daemon Driver Version: X.X
  Driver running in test mode.


  Enter command or device/request type:
```

At this point the driver will accept a device name to run a printer, punch, or Type II remote device, or a listen command to initialize a Type I remote station.

Testing a Remote Station

Assuming this remote device can accept command input (a Type I remote device), the dialog continues:

```
! listen g115_1
  Attaching line "g115_1" on channel (b.h002).
```

Responses will be different for Type II devices, but operation is essentially the same.

The test process waits here until the line becomes dialed up, and does not respond to input from the terminal; the only way to get the process's attention is to issue a quit signal. This will cause the process to print out the following message:

Enter command(early quit):

A limited set of commands is available at early quit command level, one of which is a help command which lists the few commands acceptable at this command level. The process continues waiting for the dialup event from the FNP when the user issues the start command.

When the dialup event occurs, the following message is printed:

Requesting station identifier on line "g115_1".

At the same time the message "Enter station command:" is sent to the remote device. The station command must then be entered from the remote device.

After the station command has been given, the process may be run as a normal driver process. However, because the test entry was used, several other commands have been made available to the user. One of these is the debug command. This simply calls the system debug command. From within the debug command, the user may use all the the debug command requests, including ".." to execute normal Multics commands.

Within the coordinator/driver test process there exist two pseudo processes stacked above the original interactive process: the coordinator in the middle, and the driver on top. The user's terminal communicates with the driver process after typing in "driver" during initialization. If the user issues the logout command, he logs out only the driver part of the test process; the terminal is then communicating with the coordinator part of the test process. The user may now start a new driver servicing the same or another device defined in the test directory's iod_tables. To terminate the test session, the user issues the logout command again, and the coordinator part of the process logs out. The user is now back to normal Multics interactive command level.

Setting Breakpoints

The user may wish to set breaks in the software to investigate a problem. A copy of the desired segment must be created and initiated in the test directory. If the segment normally exists in a bound object segment, all components must exist and be initiated in the test directory. At the user's option, the source can be copied into the test directory and recompiled with the map and table options. This allows full use of either the probe or the debug command to investigate the problem.

If the debug command is used to set breaks, the user should enter debug, set the breaks, and then bring up the test driver from within debug. This way the process will transfer directly to debug whenever a break point is reached.

If the probe command is used, the user may enter probe, set breaks, and optionally bring up the test driver within probe. If the test driver is already initialized, the debug command must be given in order to enter probe (via the debug request "..probe") to manipulate breaks previously set up by probe, if the process is not stopped at a probe break.

Some errors occurring before full driver initialization invoke debug automatically, while in test mode. The state of the process can be examined at this point. A ".q" debug request will perform the equivalent of a start command.

Command Level Messages

The standard command level message for the daemon coordinator/driver is:

Enter command:

Other possible levels can be:

early quit
quit
request
iodd signal (test mode only)

and are indicated parenthetically in the command level message. For example:

Enter command(quit):

SAMPLE EXEC COM FILE

The following is a sample of an exec_com that has proven useful in setting up and running a test environment. When creating your own exec_com, remember to replace TEST_DIR with the absolute pathname of the test directory.

```
&command_line off
&goto &ec_name

&label setup_environment
sa TEST_DIR>** sma [user name].[user project]
sa TEST_DIR>coord_dir>** rw [user name].[user project]
sa TEST_DIR>coord_lock rw
sa TEST_DIR>iodc_data rw
mssa TEST_DIR>([segs *.ms]) adros [user name].[user project]
& Initiate software in test directory at this point.
& set_ttt_path TEST_DIR>TTF.ttt
&quit

&label start_iod
&attach
test_io_daemon -dr TEST_DIR
coord
driver
&detach
&quit

&label use_test_queues
& Call the test entry of the daemon request commands.
dprint_$test TEST_DIR
ldr$test_ldr TEST_DIR
cdr$test_cdr TEST_DIR
&quit

&label use_system_queues
dprint_$test >ddd>idd
ldr$test_ldr >ddd>idd
cdr$test_cdr >ddd>idd
&quit

&label make_tables
& Compile the iod_tables and generate any missing message segments.
iodtc iod_tables
create_daemon_queues -dr TEST_DIR
&quit
```

TEST MODE COMMANDS

The following is a description of the test mode commands. They may be entered from the master terminal only.

Name: coord

The coord command to the driver allows the coordinator part of the test process to come to command level.

Usage

coord

Notes

This command should be followed by the start command as soon as the user is finished with coordinator command level.

The driver part of the process has not been released but suspended. The slave terminal cannot be used for input output in this condition. To reactivate the driver, use the start command.

The driver command to the coordinator is not accepted after the coord command has been given from the driver because the driver part of the process has not been released.

If a return or logout command is issued, the entire coordinator/driver test environment is released and the process returns to the original process command level.

Name: debug

The debug command calls the system debug command to allow the user to set and reset break points, execute interactive Multics commands, etc. This command is available from coordinator command level or driver command level.

Usage

debug

Notes

The driver will respond with "Calling debug" on the master terminal.

Name: driver

The driver command to the coordinator creates the driver part of the user's test process on top of the coordinator part.

Usage

driver

Notes

The driver command is accepted by the coordinator part of the user's test process only if the driver part has not been suspended previously by the coord command.

Name: pi

The pi command to the driver generates a program_interrupt signal.

Usage

pi

Notes

This allows the user to discard any undesirable output (or occurrence) by generating a quit signal, and to then return to the last stack frame with a program_interrupt handler (i.e., debug or probe). Normally, this command is used to return to the debug command when one of its functions was interrupted by a quit signal.

Name: resume

The resume command directs the driver to attempt recovery from iodd signal command level or return to normal command level from request or quit command level (aborting any current request), as if it were not in test mode.

Usage

resume

Notes

In test mode, the driver will not attempt recovery of error conditions. Instead, after all the error messages are displayed, it will stop at iodd signal command level.

Name: return

The return command to the driver does the same thing as the logout command, except that no messages are displayed, and the coordinator is not notified that the driver has logged out.

Usage

```
return
```

Note

The return command to the coordinator is the logout command.

APPENDIX E

DRIVER TO DRIVER MESSAGE FACILITY

To send a message to another user, the first user must provide enough information to uniquely identify the second user's mailbox. With the standard send_message command this is accomplished by specifying the user_id of the person to whom a message is being sent. Because most standard drivers at a given site run simultaneously as the pseudo-user IO.SysDaemon, the user_id must be replaced by identification specific to individual drivers in order to enable driver to driver communication to take place.

The unique attribute of a standard driver is the major device it is using. For a remote driver, the station_id is unique. Therefore to establish driver to driver communication, mailboxes of form <device>.mbx must be created for each standard driver and <station>.mbx for each remote driver. The mailboxes are located in the directory >daemon_dir_dir>io_msg_dir. The site administrator sets up these mailboxes by typing:

```
change_wdir >daemon_dir_dir
create_dir io_msg_dir -access_class system_low
set_acl io_msg_dir s *
change_wdir io_msg_dir
mbx_create (device1 device2 station1 station2 ...)
mbx_set_acl * adrosw *.SysDaemon adrosw *.Driver_Projects
```

The message facility can be extended to all processes by adding the extended ACL term aosw *.*.* to each mailbox (this would enable a user process to supply the device operator with a request_id and ask for that request to be run next).

At this point drivers are able to send messages to specific devices by a command line of the form:

```
send_message -pn >ddd>io_msg_dir><device> <message>
```

The next step in enabling the message facility is to define commands that allow drivers to communicate with each other. To do this the site administrator edits either the default iod_admin.ec (see Appendix C) or the device or station specific exec_coms to produce the three new driver commands x am, x sm, and x pm.

To allow drivers to accept messages the site administrator adds the following to the admin exec_com:

```
&label am.command
&
& for:  x am   -no args-
&
am -pn >ddd>io_msg_dir>[iod_val station_id] -print -call iod_driver_message
defer_messages -pn >ddd>io_msg_dir>[iod_val station_id]
&quit
```

This initializes the mailbox; the driver can then receive messages. The iod_val active function returns the major device (or station_id for remote device) that was established during driver initialization. Messages are deferred so that a remote site that relies on one printer for both listings and messages will not get messages in the middle of printing a request; for remote sites that do not direct messages to the printer (i.e., that use a separate console for slave or control output), it is also possible to remove the defer_messages command from the exec_com. The iod_driver_message program ensures that messages get to a slave if there is one active.

To allow one driver to send messages to another driver, the site administrator adds the following to the admin exec_com:

```
&label sm.command
&
& for: x sm <station> <message>
&
&if [not [exists argument &2]]
&then &goto missing_arg.error
&if [not [exists argument &3]]
&then &goto conversational_sm
sm -pn >ddd>io_msg_dir>&2 from driver [iod_val station_id]: &f3
&quit
&label conversational_sm
&print Enter your station_id as the first message line.
&print Type "." to exit send message.
send_message -pn >ddd>io_msg_dir>&2
&quit
```

To allow a driver to print any pending messages (assuming that they are deferred as shown above), the following should be added to the admin exec_com:

```
&label pm.command
&
& for: x pm   -no args needed-
&
pm -pn >ddd>io_msg_dir>[iod_val station_id] -call iod_driver_message
&quit
```

The setup for the driver to driver message facility is now complete. Each remote station operator can check for pending messages between requests by giving the x pm command. If messages have not been deferred by the x am command, each message will appear as soon as received.

APPENDIX F

IO MODULES FOR REMOTE STATIONS

hasp_workstation_ I/O Module

The hasp_workstation_ I/O module allows one or more I/O daemon processes to control the devices attached to a remote HASP workstation.

Each device of the workstation should be configured as a separate Type II I/O daemon; the slave parameter of the args substatement for each driver must be given as "slave= no"; the line substatement must specify the appropriate subchannel of the HASP multiplexed channel on which the remote station will be connected.

In this release, all commands needed to control the I/O daemons driving the devices of a HASP workstation must be entered by the central system operator. This restriction will be removed in a future release.

tty_printer_ I/O Module

The tty_printer_ module will allow the I/O daemon to run a polled VIP7760 hardcopy terminal as a printer for dprint requests. It is used with the I/O Daemon running the remote_driver_ module for type II Stations (refer to Section 2). A typical iod_table definition for a polled VIP Station would look like:

```
Device:             vip1;
  driver_module:    remote_driver_;
  line:             b.h006.p01; /* MUX channel for printer */
  args:             "station= vip1,
                    desc= -terminal tty_printer_ -comm tty_
                    -pll 118 -ttp VIP7714 -htab -vtab";
minor_device:       prt;
  minor_args:       "dev= printer";
  default_type:     vip1_prt;
```

The hardcopy device of a polled VIP station is typically an OEM version of a TN1200 with no keyboard. This means that the usable line length is 118 characters. If this line is to be set correctly during driver initialization, the request type definition in the iod_tables should specify an "rqti_seg" which sets the physical line length to 118 (otherwise, remote_driver_ will use a default line length of 132 for no rqti segment). The paper_info driver command (see Section 2) can be used to correct a bad line or page length setting.

If the hardcopy device uses a VFU tape or wheel for Form Feed (FF) and Vertical Tab (VT) control, set the stops as follows:

VT and FF at Line 1
VT only at Lines 11, 21, 31, 41, 51, 61
(assumes 66 lines per physical page)

It is always assumed that the terminal will support FF control characters.

Adding the -vtab option to the attach description of the args keyword will enable vertical tabs to be sent whenever it is more efficient than multiple New Line characters. Some terminals do a top of form function for both FF and VT control characters. Hence, the -vtab option should not be used for these terminals.

Adding the -htab option to the attach description of the args keyword causes Horizontal Tab (HT) characters to be sent instead of multiple space characters whenever possible.

Often, head and tail sheet banners and separator bars are not needed at a polled VIP station (they take a long time to print.) These can be suppressed in the rqti_segment or by the driver commands:

```
banner_type  none
banner_bars  none
```

The tty_printer_ I/O Module can also be used to make the login terminal of a non-system driver act as a printer. This would be specified as follows:

```
Device:            my_prt;
  driver_module    remote_driver_;
  line:            user_i/o;
  args:            "station= my_prt,
                    desc= -terminal tty_printer_ -comm syn_
                    -inhibit close -pll 118 -htab -vtab";

minor_device:       prt;
  minor_args:       "dev= printer";
  default_type:     private_rqt;

Request_type:       private_rqt;
  driver_userid     Person_a.Project_b;
  rqti_seg:         private_rqt_info;
  accounting:       nothing; /* use the nothing command */
  device:           my_prt.prt;
```

This will cause the printer output switch to be connected via syn_ to the user_i/o switch. The -inhibit close attach option is used to prevent driver commands, which result in device detachment, from detaching the user_i/o switch.

If the "station=" key is used in an args substatement the driver will accept any value (other than blank) and accept all minor devices to be used with all devices that dial in without any authentication controls. This would be used for a terminal without a reader (print/punch only) or with a dedicated phone line.

slave= <yes_or_no>
The "slave=" key value of "yes" is used to tell the driver that it should accept commands from the remote terminal as a slave terminal, as well as from the central site terminal (master terminal). This key is optional and is only used in the "args:" subheader of the major device. The default is no.

terminal= <terminal_type_module_name>
The "terminal=" key is used to specify which terminal module will be used for the device. This key is required (unless the "termcomm=" key is used elsewhere in an args substatement) and is only used for a major device. (For example, "ibm2780_" is a terminal module.)

<pathname>
If the first character of the args substatement is ">", the remote_driver_ assumes that the entire args string is a full pathname. The driver will look in that segment for the args string it will use. This is a useful feature if the desired args string is longer than 256 characters (the limit for all args strings in the iod_tables segment). The format is a sequence of key-value pairs separated by a comma. There may be multiple lines, but each line must end in a comma (even the last one). The entire string may be quoted as long as there is a comma before the last quote. One such segment may be used for many devices; however, there can only be one args string per segment.

APPENDIX G

THE HASP WORKSTATION SIMULATOR

Multics provides a facility for the simulation of a remote job entry (RJE) workstation using the HASP communications protocol. Through this facility, Multics users can request that job decks be transmitted to a remote system for execution and the resulting output be returned to Multics for printing/punching or online perusal.

A HASP workstation is composed of card readers, card punches, line printers, and an operator's console. Each device to be simulated by Multics is configured as a separate sub-channel of a physical communications channel defined in the CMF as a HASP multiplexer channel. (See MAM Communications for details on configuring a HASP multiplexer.) Up to eight card readers may be configured in a workstation; a total of no more than eight line printers and card punches may be configured; exactly one operator's console must be configured.

SIMULATOR STRUCTURE

The I/O daemon driver module hasp_ws_sim_driver_ simulates the operation of a workstation's card readers, line printers, and card punches; the command hasp_host_operators_console simulates the console. A separate process is used to simulate each device to permit all devices to operate asynchronously, thus achieving maximum throughput over the communications line.

The simulated operator's console is used to establish the identity of the workstation with the remote system. Subsequently, it may be used to control the operation of the workstation, request status on jobs executing on the remote system, and examine the queues of output files waiting for transmission to Multics.

Card decks are transmitted from Multics through the simulated card readers to the remote system. These decks are normally jobs to be executed by the remote system. On Multics, each card deck must be contained in a segment. A Multics user requests that a deck be transmitted by issuing the dpunch command; a separate request type is used for each remote system.

The remote system transmits output files to Multics through the simulated line printers and card punches. By default, the simulator automatically issues dprint or dpunch requests for these files as appropriate. However, a site may choose to have these output files placed into the system pool storage for subsequent retrieval by Multics users. To use this option, the driver process must be instructed to to expect control records in each output file and the remote system must include these Multics control records to indicate which Multics user owns the file. Adding control records to an output file may involve modifications to the remote computer's operating system, the JCL of each job submitted for remote execution, the programs executed by the each job, or a combination of the above. (See MPM Reference for a description of the format of these control records.)

DEFINITION OF A HASP WORKSTATION SIMULATOR

To define a workstation simulator, the local administrator(s) must:

- Define the configuration of the workstation being simulated: the number of card readers, line printers, and card punches must be agreed upon with the remote system's administrator(s).

- Determine if the remote system requires that a SIGNON control record be transmitted to establish the identity of the workstation. (The SIGNON record is a special record defined by the HASP protocol to enable the host system to establish the identity of the workstation. Many operating systems do not require this control record, but validate the workstation in other ways.) If a SIGNON record is required, it's exact content must be determined for use in the attach descriptions described below.

- Define the HASP multiplexer channel as described in MAM Communications.

- Define a major device for each simulated device except the operator's console and a request type for the submission of card decks in the system iod_tables.

- Create an ACS segment for each sub-channel of the HASP multiplexer channel, give the process which will attach that sub-channel rw access to the ACS and the dialok attribute in the PDT. (See MAM Communications and MAM System.) It is recommended that the process which attaches the simulated operator's console not be registered on the SysDaemon project.

- Determine the printer channel stops used in output files returned from the remote system and insure that the Multics request type(s) used to print those files include the appropriate logical channel stops in their RQTI segments. (See "Request Type Info Segments" in section 2 of this manual.) For example, many systems use channel stop #1 to represent the top of a page; the RQTI segments should specify "Line (1): 1;" to insure correctly formatted output.

IOD TABLES

With the exception of the operator's console, each simulated device is controlled by an I/O daemon using the hasp_ws_sim_driver_ module. A separate major device with exactly one minor device must be defined in the iod_tables for each simulated device.

The major device definition must include a line statement specifying the sub-channel of the simulated device; the "line: variable;" construct is not allowed. Additionally, an args statement must be included specifying a station ID and use of the hasp_host_ terminal I/O module (see MPM Communications).

The minor device specification must include a minor_args statement which specifies the type of device being simulated. Additional keywords may be used in this statement as described below.

See "I/O Daemon Tables" in section 2 of this manual for a description of the iod_tables source language.

Sample iod tables Definition

The iod_tables entries to simulate a HASP workstation with a card reader, card punch, and two line printers follows:

```
Device:                 cdc_rdr1;                    /* Card reader */
  line:                 a.h014.rdr1;
  driver_module:        hasp_ws_sim_driver_;
  args:                 "station= CDC, desc= -terminal hasp_host_ -comm hasp";
minor_device:           rdr1;
  minor_args:           "dev= reader_out";
  default_type:         cdc_jobs;

Device:                 cdc_prt1;                    /* Line printer #1 */
  line:                 a.h014.prt1;
  driver_module:        hasp_ws_sim_driver_;
  args:                 "station= CDC, desc= -terminal hasp_host_ -comm hasp";
minor_device:           prt1;
  minor_args:           "dev= printer_in, request_type= cdc_output";
  default_type:         dummy;

Device:                 cdc_prt2;                    /* Line printer #2 */
  line:                 a.h014.prt2;
  driver_module:        hasp_ws_sim_driver_;
  args:                 "station= CDC, desc= -terminal hasp_host_ -comm hasp";
minor_device:           prt2;
  minor_args:           "dev= printer_in, auto_queue= no";
  default_type:         dummy;

Device:                 cdc_pun1;                    /* Card punch */
  line:                 a.h014.pun1;
  driver_module:        hasp_ws_sim_driver_;
  args:                 "station= CDC, desc= -terminal hasp_host_ -comm hasp";
minor_device:           pun1;
  minor_args:           "dev= punch_in";
  default_type:         dummy;

Request_type:           cdc_jobs;            /* Request type for submitting card */
  generic_type:         punch;               /* ... decks to remote CDC system */
  max_queues:           1;
  device:               cdc_rdr1.rdr1;

Request_type:           dummy;               /* Required by line printers and */
  generic_type:         dummy;               /* ... card punches to avoid errors */
  max_queues:           1;                   /* ... from iod_tables_compiler */
  device:               cdc_prt1.prt1;
  device:               cdc_prt2.prt2;
  device;               cdc_pun1.pun1;
```

args Statement Keywords

station= <station_id>
: identifies returned output files when said files are printed/punched automatically. This keyword is required; the same value should be used for all devices of a workstation simulator.

desc= <attach_description>
specifies the attach description used to attach the terminal/device I/O module. This keyword is required. The attach description must include the "-terminal hasp_host_" and "-comm hasp" options; the "-tty" option is provided automatically by the driver process. If the remote system requires a SIGNON record, the "-signon" option must be included for all devices of the workstation. (See MPM Communications for a description of the hasp_host_ I/O module.)

minor args Statement Keywords

dev= <device_type>
specifies the type of device being simulated by this driver process. This keyword is required. The acceptable values for device_type are:

reader_out
simulates a card reader for sending card decks to the remote system.

printer_in
simulates a line printer for receiving output files from the remote system.

punch_in
simulates a card punch for receiving card decks from the remote system.

auto_receive= <switch_value>
specifies the mode of operation of this driver whenever communication is established with the remote system. The possible choices are (1) to automatically wait for output files from the remote system or (2) to listen for I/O daemon commands from the operator. The possible values for switch_value are:

yes
automatically wait for output files from the remote system whenever communication is established. (This mode is especially useful with hardwired connections.)

no
listen for I/O daemon commands whenever communication is established.

This keyword cannot be given if "dev= reader_out" is specified. This keyword is optional; the default value is "no" (listen for I/O daemon commands).

auto_queue= <switch_value>
specifies whether output files received by this driver are (1) automatically printed or punched locally or (2) scanned for Multics control records and made available for online perusal as described above. The possible values for switch_value are:

yes
automatically queue the files for printing/punching; do not scan for control records, or

no
scan the output files for Multics control records and store them in system pool storage for online perusal; do not automatically queue files for printing/punching.

This keyword cannot be given if "dev= reader_out" is specified. This keyword is optional; the default value is "yes" (automatically queue output files).

request_type= <rqt_name>
rqt= <rqt_name>
specifies the Multics request type to be used for automatically printing or punching output files. The request type specified must be of generic type "printer" if "dev= printer_in" is given or generic type "punch" if "dev= punch_in" is given; this keyword cannot be given if "dev= reader_out" is specified. This keyword is optional; the default request type used is the default specified for the appropriate generic type.

This page intentionally left blank.

OPERATING A HASP WORKSTATION SIMULATOR

SIMULATOR INITIALIZATION

To start a HASP workstation simulator:

- If necessary, issue the initializer "load_mpx" command described in the MOH to cause the HASP multiplexer channel to wait for a connection.

- Login the process which is to run the simulated operator's console of the workstation and issue the hasp_host_operators_console (hhoc) command, described below, to wait for the connection to be completed. If the remote system requires a SIGNON record as part of the connection procedure, include the "-signon" option on the hhoc command line.

- Complete the physical connection to the remote system.

- When the process running the operator's console prints the message "Input:"indicating that the physical connection is established, perform any logon sequence required to identify the workstation to the remote system. The exact sequence used, if any, should be determined from the remote system's administrative staff.

- Login each of the driver processes for the other simulated devices. The sequence used to login a driver process is described in "Login and Initialization of Device Drivers" in section 3 of this manual.

- On the terminal of the process running the operator's console, issue any commands to the remote system required to ready all the devices of the workstation.

- For each driver process running a simulated card reader, issue the commands:

  ```
  ready
  pun_control autopunch
  go
  ```

  These commands will start the transmission of card decks to the remote system.

- Issue the "receive" command for each driver process running a simulated line printer or card punch. This command will cause these drivers to wait for output files to be sent by the remote system. As each output file is received, it is processed according to the specifications given in the minor_args statement of the driver as described above.

SPECIAL INSTRUCTIONS FOR RUNNING THE PRINTER AND PUNCH SIMULATORS

In addition to the commands described in this section, the only other I/O daemon commands which may be used in the driver process of a simulated line printer or card punch are: logout, hold, new_device, inactive_time, x, start, help, status, reinit, release, and clean_pool. These commands are described in section 3 of this manual.

After use of the "receive" command described below, the driver only recognizes pending commands while it is between output files. If it is necessary to execute a command while a file is being received, a QUIT must be issued to the driver to bring the driver to QUIT command level. The "hold" command can then be used to cause the driver to remain at QUIT level; the "release" command can be used to abort receiving the file and return to normal command level; and the "start" command can be used to resume receiving the file.

Name: receive

The receive command causes the driver to wait for output files to be transmitted from the remote system. A message is issued at the start and end of each file received. If automatic queueing of output files is enabled for this simulated device, output files will be locally printed or punched after they have been successfully received; otherwise, the output files will be placed into system pool storage as specified by the ++IDENT control records which must be present in the files.

Usage

```
receive
```

Name: auto_queue

The auto_queue command controls whether output files received by this driver are (1) automatically printed or punched locally or (2) scanned for Multics control records and placed in system pool storage for online perusal.

Usage

```
auto_queue <switch_value>
```

where:

switch_value
: must be chosen from:

  yes
  : automatically queue the files for printing/punching; do not scan for control records, or

  no
  : scan the output files for Multics control records and store them in system pool storage for online perusal; do not automatically queue files for printing/punching.

Name: request_type, rqt

The request_type command is used to specify the request type to be used for the automatic queuing of output files received by this device.

Usage

```
rqt <rqt_name>
```

where:

1. rqt_name
   is the name of the request type to be used for automatic queuing. The generic type of this request type must agree with the type of device being simulated ("printer" for simulated line printers, etc). This parameter is optional; the default value is the request type specified in the iod_tables definition of this driver.

Name: hasp_host_operators_console, hhoc

The hasp_host_operators_console command is used to simulate the operation of the operator's console of a HASP workstation. The operator's console is used to identify a workstation to a remote system, to issue commands governing the operation of the workstation, and to receive status information from the remote system.

Usage

```
hhoc tty_channel {control_arguments} {attach_arguments}
```

where:

1. tty_channel
   is the name of the terminal channel to be attached as the operator's console. This channel must be configured as the console sub-channel of a HASP multiplexer channel (eg: a.h014.op). See MAM Communications for a further description of the HASP multiplexer.

2. control_arguments
   may be chosen from the following:

   -signon STR
   specifies that the remote host requires a SIGNON record to be transmitted before data transmission may occur. STR is the text of the control record; it may be up to 80 characters in length. Before transmission it is translated to uppercase and the remote system's character is set.

   -no_signon
   specifies that the remote host does not require a SIGNON record. (Default)

3. attach_arguments
   are options acceptable to the hasp_host_ I/O module. This command supplies the -comm, -tty, and -device options automatically; these options need not be given on the command line. (See MPM Communications for a description of the hasp_host_ I/O module.)

Notes

If the remote system requires a SIGNON, the -signon option should be supplied on the command line specifying the exact SIGNON record to be transmitted.

For example, the command line:

```
hhoc a.h014.opr -signon "/*SIGNON REMOTE7"
```

may be used to attach the channel a.h014.opr as the operator's console of a remote IBM system expecting a connection from the workstation named REMOTE7.

After attaching the channel specified on the command line, hasp_host_operators_console prompts the user for terminal input with the string "Input:".

Input from the terminal is transmitted directly to the remote system unless the line begins with the request character, an exclamation mark (!); lines beginning with the request character are interpreted by this command. The valid requests are described below.

Any text received from the remote system is displayed directly on the terminal without any interpretation by hasp_host_operators_console.

HASP HOST OPERATORS CONSOLE REQUESTS

The following requests are recognized by hasp_host_operators_console when given at the beginning of a line of terminal input:

!.. <REST_OF_LINE>
: the rest of the line is passed to the Multics command processor for execution as ordinary commands.

!.
: prints a message of the form:

    hasp_host_operators_console N.N; connected to channel NAME.

  where N.N is the current version of this program and NAME identifies the channel connected as a console to the remote system.

!quit
: causes the command to hangup the operator's console channel and return to Multics command level.

INDEX

A

abort command 4-5

access class 2-10, 2-20
  ++AIM cards 4-4
  daemon_dir_dir directory 2-1

admin exec_com C-1, E-1
  sample C-2

administrative commands A-1
  create_daemon_queues A-2, D-2
  cv_prt_rqti 2-21, A-3
  display_prt_rqti A-4
  iod_command A-4.1
  iod_tables_compiler 2-3, 2-18, A-5
  iod_val active function A-6, C-1, E-2
  print_devices 3-5, A-7
  print_iod_tables 2-19, A-8
  print_line_ids A-9
  print_spooling_tape A-10

AIM 1-1, 2-10, 2-20, 3-1.1
  access class 2-1
  features 3-1.1
  source file example 2-12

auto_print mode 3-17

auto_queue G-7

auto_start_delay command 3-25

B

banner_bars command 3-25

banner_type command 3-26

C

cancel command 3-26

card input station management 4-1

card input station management (cont)
  access control
    password 4-1
    registration 4-1
    RJE submission 4-1
    station password 4-1
    station registration 4-1
    system station access control segment 4-2, D-2
    user access segment 4-2
  command summary B-4
  reading cards at the central site 4-4
    daemon communication 4-5
      abort command 4-5
      clean_pool command 4-5
      help command 4-5
      logout command 4-5
      read_cards command 4-5
      reinit command 4-5
      start command 4-5
    error conditions 4-6
    login and initialization 4-5
  reading cards, remote site 4-6
  reading user card decks 4-2.1

Card_input.Daemon 4-1, 4-4

clean pool command 4-5

clean_pool command 3-27

control terminals 3-8
  driver initialization 3-8

coord command D-7

copy command 3-27

create_daemon_queues command A-2, D-2

ctl_term command 3-15, 3-28

cv_prt_rqti command 2-21, A-3

D

daemon
  see I/O daemon

debug command D-4, D-8

defer command 3-29

defer_time command 3-29

device classes 3-7
  driver initialization 3-4
  logging in a driver 3-4
  substatements 2-11

device drivers 3-4, E-1

device specific driver commands 3-13
  summary B-3

display_prt_rqti command A-4

driver
  command levels 3-9
    early quit D-4, D-5
    iodd signal D-5
    normal 3-10
    quit 3-10, D-5
    request 3-10, D-5
  command summary B-1
  commands 3-23
    abort 4-5
    auto_start_delay 3-25
    banner_bars 3-25
    banner_type 3-26
    cancel 3-26
    clean_pool 3-27, 4-5
    copy 3-27
    ctl_term 3-15, 3-28
    defer 3-29
    defer_time 3-29
    go 3-30
    halt 3-30
    help 3-31, 4-5
    hold 3-31
    inactive_limit 3-32
    kill 3-32
    logout 3-33, D-4
    master 3-33
    new_device 3-6, 3-34
    next 3-34
    paper_info 3-35
    pause_time 3-36
    print 3-37
    prt_control 3-37
    punch 3-38
    pun_control 3-39
    ready 3-40
    read_cards 4-3, 4-5
    reinit 3-41, 4-5
    release 3-41
    req_status 3-42
    restart 3-43
    restart_q 3-44
    runout_spacing 3-44
    sample 3-46
    sample_form 3-47
    sample_hs 3-47
    save 3-48
    sep_cards 3-49
    single 3-50

driver (cont)
  commands
    slave 3-50
    slave_term 3-51
    start 3-52, 4-5
    station 2-15, 3-6, 3-52
    status 3-53
    step 3-54
    x 3-54, A-6, C-1, D-2, E-1
  initialization 3-8
  message facility E-1
  printer driver 3-16
    see printer driver
  punch driver 3-18
    see punch driver
  remote driver 3-21
    see remote driver
  spool driver 1-1, 3-18
    see spool driver
  terminal control 3-8
  test mode D-1
    commands D-7
      coord D-7
      debug D-8
      driver D-8
      pi D-9
      resume D-9
      return D-10

driver command D-8

E

early quit command level D-5

exec_com
  admin exec_com C-1, D-6, E-1
  test mode D-6

G

go command 3-30

H

halt command 3-30

HASP workstation simulator G-1, G-5
  commands
    auto_queue G-7
    hasp_host_operators_console, hhoc G-9
    receive G-6
    request_type, rqt G-8
  definition G-2
  iod_tables G-2, G-3
    sample definition G-3
  operation G-5
  simulator G-1
  special instructions, printer and punch simulators G-5

HASP workstation simulator (cont)
  statement keywords G-3, G-4

hasp_host_operators_console, hhoc G-9

help command 3-31, 4-5

hold command 3-31

I

I/O coordinator 2-1, 2-12, 2-18, 3-1
  commands 3-3
    help 3-4
    list 3-3
    logout 3-3
    print_devices 3-3
    restart_status 3-4
    term 3-4
    wait_status 3-3
  initialization 3-1
  login 3-1
  test mode D-3

I/O daemon 1-1, 2-1, 3-1
  admin_exec_com C-1
  AIM maintenance 2-20
  command summary B-1
  directories 2-1
    cards 2-3
    daemon_dir_dir 2-1
    io_daemon_dir 2-1, D-1
    io_msg_dir 2-3
  queues 2-19
  request type info segment
    see rqti segment
  search rules 3-1
  tables 2-1, 2-3, 2-12, 2-15, D-1,
      G-3
    creation and maintenance 2-18
    simulated device G-2
    source language 2-3
      AIM features 2-10
      major and minor devices 2-8
      source file example 2-7
      source file example, AIM 2-12
      source file example, minor
          devices 2-9
      statements 2-4
      substatement, default request
          2-12
      substatements for devices 2-5
      substatements for lines 2-4
      substatements for minor devices
          2-9
      substatements for request types
          2-6
      substatements, device classes
          2-11
      syntax 2-3
    standard Driver
      command summary 3-10
      modules 2-13

inactive_limit command 3-32

IO modules for remote stations F-1
  remote_driver F-1
  tty_printer F-1

IO.SysDaemon 2-8, 3-2

iodd signal command level D-5

iod_admin.ec
  see admin_exec_com

iod_command command A-4.1

iod_tables segment
  see I/O daemon tables

iod_tables_compiler command 2-3, 2-18,
    A-5

iod_val active function A-6, C-1, E-2

K

kill command 3-32

L

logout command 3-33, 4-5, D-4

M

major device 2-8

master command 3-33

master terminal 3-8

message facility E-1

message segments 2-19

minor device 2-8, 3-5
  source file example 2-9
  substatements 2-9

multifunction device 3-5
  card reader 3-14

N

new_device command 3-6, 3-34

next command 3-34

P

paper_info command 3-35

pause_time command 3-36

pi command D-9

preprinted accountability forms 3-14

print command 3-37

printer driver 2-14, 3-16
  command summary B-3
  request command level 3-10

print_devices command 3-5, A-7

print_iod_tables command 2-19, A-8

print_line_ids command A-9

print_spooling_tape command A-10

probe command D-4

prt_control command 3-37

punch command 3-38

punch driver 2-14, 3-18

pun_control command 3-39

Q

queues 2-19

quit command level 3-10, D-5

R

reader_driver 2-14

ready command 3-40

read_cards 4-5

read_cards command 4-3

receive G-6

reinit command 3-41, 4-5

release command 3-41

remote driver 2-15, 3-21, 4-2.1
  arguments 2-17
  command summary B-4.1
  initializing 3-22
  Type I stations 2-15, 2-17, D-3
  Type II stations 2-17, 2-18, D-3

request command level 3-10, 3-17, D-5

request type info (rqti) segment
  see rqti segment

request_type, rqt G-8

req_status command 3-42

restart command 3-43

restart_q command 3-44

resume command D-9

return command D-10

rqti segment 2-7, 2-20, D-1
  source segment example 2-24
  source segment syntax 2-21

runout_spacing command 3-44

S

sample command 3-46

sample_form command 3-47

sample_hs command 3-47

save command 3-48

sep_cards command 3-49

single command 3-50

slave command 3-50

slave terminal 2-17

slave_term command 3-51

spool driver 1-1, 2-14.1, 3-18
  command summary B-4.1
  commands 3-21
  messages 3-21
  tape description A-11

standard driver
  command summary 3-10
  modules 2-13

start command 3-52, 4-5

station command 2-15, 3-6, 3-52

status command 3-53

step command 3-54

T

terminals 3-8

X

x command 3-54, A-6, C-1, D-2, E-1

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 68) MULTICS BULK INPUT/OUTPUT	ORDER NO.	CC34, REV. 1
		DATED	MARCH 1979

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

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

FROM: NAME ____________________ **DATE** ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

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

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 68)
MULTICS BULK INPUT/OUTPUT
ADDENDUM A

ORDER NO. CC34-01A

DATED DECEMBER 1979

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE ____________

TITLE ______________________

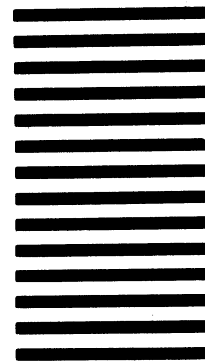
COMPANY ______________________

ADDRESS ______________________

____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 68)
MULTICS BULK INPUT/OUTPUT
ADDENDUM B

ORDER NO. CC34-01B

DATED FEBRUARY 1980

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE ____________

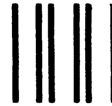
TITLE ______________________

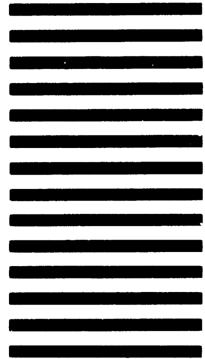
COMPANY ______________________

ADDRESS ______________________

____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms





BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

- - CUT ALONG LINE - -

TITLE	LEVEL 68 MULTICS BULK INPUT/OUTPUT ADDENDUM C	ORDER NO.	CC34-01C
		DATED	JULY 1981

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

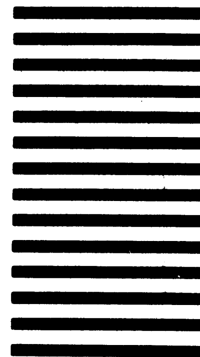
____

CUT ALONG LINE

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	MULTICS BULK INPUT/OUTPUT ADDENDUM D	ORDER NO.	CC34-01D
		DATED	JULY 1982

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

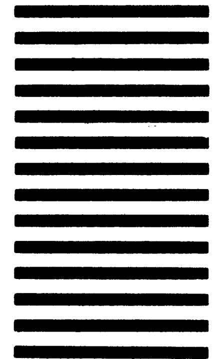
CUT ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154



POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

SERIES 60 (LEVEL 68)
MULTICS BULK INPUT/OUTPUT ADDENDUM A

SUBJECT

Information Needed by System Administrators and Operators in the Management of Bulk Input/Output

SPECIAL INSTRUCTIONS

This is the first addendum to CC34, Revision 1, dated December 1978.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Throughout the manual, change bars in the margin indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of this manual.

Three new commands for punches: punch, pun_control, and sep_cards have been added to Section 3. Also, Appendix F from Revision 1 has been deleted and has been replaced with an all new Appendix F, IO Modules for Remote Stations.

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

CC34-01A

December 1979

26353
7.5C1279
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

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

Remove	Insert
title page, preface	title page, preface
iii through vi	iii through vi
2-1, 2-2	2-1, 2-2 2-2.1, blank
2-15 through 2-24	2-15 through 2-18 2-18.1, blank 2-19 through 2-22 2-22.1, blank 2-23, 2-24
3-1 through 3-50	3-1 through 3-54
4-1 through 4-4	4-1, 4-2 4-2.1, blank 4-3, 4-4
A-5, A-6	A-5, A-6 A-6.1, blank
B-3, B-4	B-3, B-4 B-4.1, blank
C-1, C-2	C-1, C-2
D-1, D-2	D-1, D-2 D-2.1, blank
D-5, D-6	D-5, D-6
F-1 through F-3, blank	F-1, F-2
i-1 through i-4	i-1 through i-3, blank

© Honeywell Information Systems Inc., 1979

File No.: 1L13

12/79

CC34A

Roach

SERIES 60 (LEVEL 68)
MULTICS BULK INPUT/OUTPUT ADDENDUM B

SUBJECT

Information Needed by System Administrators and Operators in the Management of Bulk Input/Output

SPECIAL INSTRUCTIONS

This is the second addendum to CC34, Revision 1, dated December 1978.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Throughout the manual, change bars in the margin indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of this manual.

Due to a printing error in the last update of this manual, a page was omitted in Section 2; therefore we are reissuing pages 2-17 through 2-20 which will correct this error.

Note:

Insert this cover behind the manual cover to indicate the updating of this document with Addendum B.

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

CC34-01B

February 1980

Honeywell

26941
1.2380
Printed in U.S.A.

COLLATING INSTRUCTIONS

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

Remove	Insert
iii through vi	iii through vi
2-17, 2-18	2-17, blank
2-18.1, blank	2-17.1, 2-18
2-19, 2-20	2-19, 2-20

© Honeywell Information Systems Inc., 1980

Roach

LEVEL 68

MULTICS BULK INPUT/OUTPUT ADDENDUM C

SUBJECT

Information Needed by System Administrators and Operators in the Management of Bulk Input/Output

SPECIAL INSTRUCTIONS

This is the third addendum to CC34, Revision 1, dated December 1978.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Throughout the manual, change bars in the margin indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of this manual.

Note:

Insert this cover behind the manual cover to indicate the updating of this document with Addendum C.

SOFTWARE SUPPORTED

Multics Software Release 9.0

ORDER NUMBER

CC34-01C

July 1981

32238
5C881
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

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

Remove	Insert
title page, preface	title page, preface
iii through vi	iii through vi
2-5 through 2-8	2-5, 2-6 2-6.1, blank 2-7, 2-8
2-13, 2-14	2-13, 2-14 2-14.1, blank
3-1, 3-2	3-1, blank 3-1.1, 3-2
3-17, 3-18	3-17, blank 3-17.1, 3-18
3-35, 3-36	3-35, 3-36
4-5, 4-6	4-5, 4-6
A-1, A-2	A-1, A-2
D-1, D-2	D-1, D-2
D-3 through D-6	D-3 through D-6
F-1, F-2	F-1, F-2
	G-1 through G-10
i-1 through i-3, blank	i-1 through i-4

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

© Honeywell Information Systems Inc., 1981

File No.: 1L13

7/81

CC34C

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

23098, 5C379, Printed in U.S.A.

CC34, Rev. 1