SERIES 60 (LEVEL 68)

MULTICS COMMUNICATION SYSTEM
SYSTEM DESIGNERS' NOTEBOOK

SUBJECT

Description of the Multics Communication System

SPECIAL INSTRUCTIONS

This System Designers' Notebook (SDN) completely supersedes AN85, Rev. 0. The manual has been extensively revised to reflect software changes incorporated in the current Multics Software Release, and therefore does not include change indicators (asterisks or change bars).

This SDN describes certain internal modules constituting the Multics System. It is intended as a reference for those who are thoroughly familiar with the implementation details of the Multics operating system. Interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only.

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent SDN updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

SOFTWARE SUPPORTED

Multics Software Release 7.0

ORDER NUMBER

AN85-01

October 1979

Honeywell

Preface

Multics manuals are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The manuals contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

This manual contains a description of the software that makes up the Multics Communication System. This description is by no means complete in all its details; for a thorough understanding of Multics Communication System, or of any particular area within this system, this manual should be used for reference in conjunction with the source of the relevant programs.

Sites that intend to add specialized communication protocols to the system, or to modify existing protocols for special application, should pay particular attention to Section 12, which includes a description of the macro language in which FNP control tables are written and instructions for incorporating a new set of control tables into the FNP software.

© Honeywell Information Systems Inc., 1979

File No.: 1L13

In addition to this manual, the volumes of the Multics Programmers' Manual (MPM) should be referred to for details of software concepts and organization, external interfaces, and for specific usage of Multics Commands and subroutines. These volumes are:

MPM Reference Guide, Order No. AG91

MPM Commands and Active Functions, Order No. AG92

MPM Subroutines, Order No. AG93

MPM Subsystem Writers' Guide, Order No. AK92

MPM Peripheral Input/Output, Order No. AX49

MPM Communications Input/Output, Order No. CC92

CONTENTS

		Page
Section 1	Overview of Ring 0 Multics Communication System	1-1
	Introduction	1-1
	Responsibilities of the CS	1-1
	Structure of the CS Portion of Multics Communication System	1-2
	FNP Interface Modules	1-2
	User Interface Modules	1-3
	tty_write Module	1-3
	tty_read Module	1-3
	tty_index Module	1-3
	Multiplexing Interfaces	1-4
	Some Standard Sequences	1-4
	Channel Initialization	1-4
	Dialup	1-4
	Output	1-5
	Input	1-5
	Quit	1-5
	Hangup	1-6
Section 2	Hardcore Data Bases	2-1
	Data Bases Used for Communication with FNP	2-1
	dn355_data	2-1
	dn355_mailbox	2-1
	tty_buf	2-2
	tty_buf Header	2-3
	Circular Buffer	2-3
	Logical Channel Table (LCT)	2-3
	Dynamic Portion of tty_buf	2-4
	Free Block	2-4
	Data Buffer	2-5
	Per-Channel Control Blocks	2-5
	Wired Terminal Control Block (WTCB)	2-5
	Physical Channel Block (PCB)	2-6
	Pseudo-DCWs	2-6
	Delay Queue Entries	2-6
	tty_area	2-6
	Logical Channel Name Table (LCNT)	2-7
	Terminal Control Block (TCB)	2-7
	tty_tables	2-7

CONTENTS (cont)

		Page
Section 3	Multiplexing.	3-1
	Multiplexed Channels	3-1
	Basic Design.	3-1
	Multiplexer Types.	3-2
	Calls and Interrupts	3-3
	channel_manager AND priv_channel_manager.	3-3
	Entries to channel_manager.	3-4
	Interrupt Types and Associated Data.	3-11
	Entries in priv_channel_manager . .	3-14
	Multiplexer Transfer Vector -- cmtv. .	3-20
	Programming Considerations -- Wired Code.	3-21
	Multiplexer Initialization	3-21
	Database Initialization	3-21
	Loading	3-22
	Subchannel Initialization	3-22
	Crashing and Reinitialization.	3-22
	Examples of Call and Interrupt Paths .	3-23
Section 4	FNP Interface Module.	4-1
	Calls and Interrupts	4-1
	Mailbox Transactions	4-2
	Transactions Initiated by the FNP .	4-2
	FNP-Controlled Mailboxes	4-2
	Processing the RCD	4-2
	send_output Operation Code . . .	4-3
	Short Input -- input_in_mailbox.	4-3
	Long Input -- accept_direct_input	4-4
	Handling the accept_input Interrupt	4-4
	Transactions Initiated by the CS. .	4-5
	Output Data.	4-5
	Global Operations.	4-6
	Locks.	4-6
	Delay Queuing.	4-7
Section 5	Interfaces to the User Ring	5-1
	Output Conversion - tty_write.	5-2
	Preliminary Conversion.	5-3
	Formatting.	5-4
	Translation	5-5
	Buffer Allocation and Copying . . .	5-5
	Space Allocation and Character Counting	5-6
	Input Conversion - tty_read.	5-7
	Space Management.	5-9
	Copying	5-10

CONTENTS (cont)

		Page
	In 'rawi' Mode	5-10
	Not In 'rawi' Mode	5-10
	Translation	5-11
	Canonicalization.	5-11
	Erase and Kill Processing	5-13
	Escape Sequence Processing.	5-14
	Echo Negotiation.	5-16
	Utility Functions - tty_index.	5-17
	Initializing a Channel.	5-17
	Terminating a Channel	5-17
	Assigning a Channel to a Process. .	5-17
	Assigning an Event Channel.	5-18
	Separating a Channel from a Process	5-18
	Ascertaining the State of a Channel	5-19
	Aborting Input and/or Output. . . .	5-19
	Control Operations.	5-19
	read_status Operation.	5-20
	write_status Operation	5-20
	printer_off Operation.	5-20
	printer_on Operation	5-21
	set_terminal_data Operation. . .	5-21
	wru Operation.	5-22
	Modes.	5-22
	Privileged Operations	5-23
	dump_fnp Operation	5-23
	patch_fnp Operation.	5-23
	fnp_break Operation.	5-24
	enable_breakall_mode Operation .	5-24
	disable_breakall_mode Operation.	5-24
Section 6	Hardcore Utilities.	6-1
	Locking and Queuing.	6-1
	Space Management	6-3
	Allocation.	6-3
	Freeing	6-4
	Needed Space.	6-4
	Assembler Language Utilities	6-5
	tty_util_ Module.	6-5
	dn355_util Subroutine	6-5
Section 7	Initialization of Hardcore Data Bases . .	7-1
	Preliminary Initialization	7-1
	LCT Initialization	7-2
	Multiplexer Initialization	7-2
	Channel Initialization	7-2
Section 8	Tools and Debugging Aids.	8-1
	Tools.	8-1
	tty_meters.	8-1
	tty_dump.	8-2

CONTENTS (cont)

		Page
	tty_analyze	8-4
	Syserr Messages.	8-6
	FNP Crashes	8-6
	Other FNP Messages.	8-6
	Crashes Generated by Multics Communication System.	8-6
	Lock Errors	8-7
	Free Space Errors	8-7
Section 9	Overview of the FNP Software.	9-1
	Responsibilities of the FNP.	9-1
	Structure of FNP Multics Communication System.	9-1
	Data Bases.	9-3
	Channel Management	9-3
	Interrupts and Scheduling.	9-3
	Data Paths	9-4
	Input	9-4
	Output.	9-4
Section 10	FNP Data Bases.	10-1
	System Communications Region	10-1
	IOM Table.	10-1
	Hardware Communications Regions. . . .	10-3
	Software Communications Regions. . . .	10-3
	LSLA Software Communications Region	10-4
	HSLA Software Communications Region	10-4
	LSLA Table	10-4
	HSLA Table	10-5
	Terminal Information Block	10-7
	TIB Table.	10-7
	Buffer Pool.	10-7
	Free Block.	10-7
	Input/Output Buffer	10-7
	Echo Buffer	10-8
	DIA Request Queue	10-8
	Error Message Queue	10-9
	Delay Timing Tables	10-9
	Other Blocks.	10-10
Section 11	Scheduler	11-1
	Master Dispatcher.	11-1
	Secondary Dispatcher	11-3
	Timer Management	11-4
	Elapsed Time Metering.	11-7
	Idle Time Metering.	11-7
	Instruction Counter Sampling. . . .	11-7

CONTENTS (cont)

Page

Section 12 Terminal And Line Control 12-1
Organization of the Control Tables . . 12-1
Division Into Modules 12-1
Tables Included in the control_tables Module. 12-2
Header 12-2
Device Info Table. 12-3
Device-Type/Speed Table. 12-5
Addressing Strings 12-5
Control Table Interpreter. 12-5
Timeout 12-6
Output. 12-6
Test-state. 12-6
Status. 12-6
Status And Control Bits. 12-7
Status Bits 12-7
Control Bits. 12-9
First Word 12-9
Second Word. 12-10
Op Blocks. 12-10
TIB Extension Addressing. 12-10
Op Block Summary Lists. 12-11
Description of Scan Control Strings 12-39
Providing Additional Control Tables. . 12-43
Pseudo-ops and Data-defining Macros 12-44
Listing Controls 12-44
External Symbol Definitions. . . 12-44
Internal Symbol Definitions. . . 12-45
Character Control Tables 12-45
Suppression of Op Block Expansion 12-46
Interaction with the Main Control Tables Module. 12-46
Line Types 12-46
Answerback Reading 12-47
Useful Labels in the Main Module 12-47
Programming Considerations. 12-47
Requests from the CS 12-48
Output from the CS 12-48
Input from the Channel 12-48
Abnormal Conditions on the Channel 12-49
Line Control and Line Status . . 12-49
User-Ring I/O Modules. 12-50
Example of a Control Tables Module. 12-50

Section 13 FNP Hardware Managers 13-1
DIA. 13-1
Operation of the DIA. 13-1
DIA Transactions. 13-3

CONTENTS (cont)

Page

Queues 13-4
Interrupt Handlers. 13-5
Summaries of DIA Transactions . . . 13-5
Transactions Initiated by the CS 13-5
Transactions Initiated by the FNP 13-6
Input Data 13-7
LSLA 13-9
Operation of the LSLA 13-9
Interrupt Processor 13-10
Transmit Status Handling 13-10
Receive Status Handling. 13-11
Abnormal Status Handling 13-11
Output Frame Generator. 13-12
Command Sub-ops 13-13
Input Frame Processor 13-13
Echoing 13-15
HSLA 13-15
Calls to hsla_man 13-16
HSLA Status 13-17
Example of HSLA Processing. 13-17
CCT Management. 13-18
Echoing 13-19
Console. 13-20

Section 14 FNP Utility Functions 14-1
Space Management 14-1
TIB Address Calculation. 14-2
Fault Processing 14-2
IOM Channel Faults. 14-3
Metering 14-3
Output Sub-Ops 14-4
Tracing. 14-5

Section 15 Loading and Initialization. 15-1
bind_fnp Command 15-1
load_fnp_ Subroutine 15-2
gicb Routine 15-4
init Module. 15-5
DIA Initialization. 15-5
HSLA Initialization 15-6
LSLA Initialization 15-6
TIB Initialization. 15-7
Completion of Initialization. . . . 15-7
Status Reporting. 15-8

Section 16 FNP Crash Analysis. 16-1
How the FNP Crashes. 16-1
Dumping the FNP. 16-1
fdump_fnp_. 16-1

CONTENTS (cont)

Page

FD355 and DMP355. 16-2
Obtaining A Printed Dump. 16-2
Interpreting An FNP Dump 16-2
Format of the Dump. 16-3
Crash Reason. 16-4
Fault Identification. 16-5
Machine Registers 16-5
Trace Table 16-6
Tracing Subroutine Calls. 16-6
Other Useful Information. 16-7

Section 17 FNP-Related Commands. 17-1
Core Image Preparation 17-1
map355. 17-1
coreload. 17-2
Dump Analysis. 17-3

Appendix A Mailbox Operation Codes A-1
Operation Codes Sent from the CS to the FNP A-1
Operations Sent with a WCD I/O Command. A-1
Operations Sent with a WTX I/O Command. A-6
Operations Sent with an RTX I/O Command. A-6
Operation Codes Sent from the FNP to the CS. A-7
Operations Sent with an RCD I/O Command. A-7
Subtypes used with Alter Parameters Operations. A-9
Mode Changes. A-9
Other Subtypes. A-10

Appendix B Command Descriptions. B-1
coreload. B-2
debug_fnp, db_fnp B-3
Selecting debug_fnp Mode B-3
Summary of debug_fnp Requests. . B-16
online_dump_355, od_355 . . . . B-21
online_dump_fnp, od_fnp B-22
tty_analyze, tta. B-24

Appendix C FNP Memory Configurator C-1

Appendix D Layout of FNP Memory. D-1

CONTENTS (cont)

Page

Appendix E Automatic Baud Rate Detection E-1
Lead Control Selection of 1200 Baud. . E-1
Bit Sampling Selection of Other Baud Rates E-1
Modems E-2
Modem Options Needed for Autobaud . E-2
Algorithm. E-2
Notes E-3

ILLUSTRATIONS

Figure 3-1 Possible Paths of write Call. 3-24
Figure 3-2 Possible Paths of Interrupt 3-25
Figure 13-1 FNP DIA DCW Format. 13-2

SECTION 1

OVERVIEW OF RING 0 MULTICS COMMUNICATION SYSTEM

INTRODUCTION

The Multics Communication System is responsible for the transmission of characters between the Multics virtual memory and various user equipment (particularly terminals) connected by means of telecommunications channels to the Front-End Network Processor (FNP). The software constituting Multics Communication System resides partly in ring zero of the central system (CS) and partly in the FNP itself. Communication between the two computers (the CS and the FNP) takes place over the Direct Interface Adapter (DIA), a peripheral device of the FNP. A Multics CS may be connected to up to four FNPs simultaneously. The FNP for which Multics Communication System was originally designed was called the DATANET 355, which is why the strings "355" and "dn355" appear in the names of many of the programs and data bases described in this document.

This manual is intended to provide a general understanding of the workings of Multics Communication System. For complete details of implementation, the reader should examine the source code itself.

Section 1 through Section 8 of this manual describe the portion of Multics Communication System that is resident in the CS; Sections 9 through 17 describe the portion resident in the FNP. The appendices contain details of implementation such as command descriptions, specific codes used in various contexts, etc.

RESPONSIBILITIES OF THE CS

The primary responsibilities of the CS with respect to communications are in the following areas:

1. The association of communications channels (e.g., the data path to a user's terminal) with Multics processes;

2. dispatching input from the various communications channels to the appropriate processes;

3. converting output supplied by Multics processes to a form suitable for sending to its destination (e.g., terminals);

4. management of the ring 0 terminal I/O buffer space (tty_buf);

5. multiplexing and demultiplexing the subchannels of a concentrator channel;

6. management of the FNP, i.e., loading it at system initialization time, recovering when it crashes, etc.

STRUCTURE OF THE CS PORTION OF MULTICS COMMUNICATION SYSTEM

The major components of the CS portion of Multics Communication System are the FNP interface modules, three user interface modules (tty_write, tty_read, and tty_index), and interfaces between levels of multiplexing. Other components include utility routines for the management of locks and buffer space; initialization routines; and special subroutines used by the user interface modules. Brief descriptions of the major components appear below; all the components are discussed in more detail in Sections 4-8. The multiplexing mechanism is described in Section 3.

FNP Interface Modules

The standard FNP interface module is dn355; it is the only program in the system that communicates directly with the FNP (except for the routines that load and dump the FNP). If a root-level multiplexer (see Section 3) other than a DATANET 6600 or DATANET 6670 running Multics Communication System is used, a site-supplied module interfacing to this multiplexer would replace dn355. Except as otherwise noted, the discussions in this manual assume that the standard module is being used.

The dn355 module is invoked in either of two ways: it is called from tty_write and tty_index through the FNP multiplexer module, fnp_multiplexer, to send output data and control information, respectively, to the FNP; and it processes interrupts from the FNP when the latter sends input data or control information. Control information is passed between the FNP and the CS by means of a wired segment named dn355_mailbox; user data is read into and written from a wired segment named tty_buf. These two segments are described in more detail in Section 2.

In general, all I/O over the DIA is initiated by the FNP; the only kind of I/O operation initiated by dn355 is an interrupt that instructs the FNP software to read a submailbox of dn355_mailbox. The details of this mechanism are spelled out in Sections 4 and 13.

User Interface Modules

The three user interface modules, tty_write, tty_read, and tty_index, are called from the user ring through entries in the gate hcs_. User processes usually do this through the I/O switch mechanism, i.e., by calling entries in iox_ which call entries in the user-ring terminal I/O module, tty_. The answering service, which requires more direct control over the channel, calls the hcs_ entries directly. See also the User Ring Input/Output PLM, Order No. AN57.

The uses of these three modules are explained briefly below; they are described in more detail in Section 5.

tty_write MODULE

The tty_write module prepares user-supplied output for transmission to the FNP, places the output in tty_buf, and calls channel_manager$write to initiate the transmission. If there is insufficient space in tty_buf to hold all of the supplied output at once, tty_write only processes part of it; the caller generally goes blocked in this case, and receives a wakeup when Multics Communication System is ready to handle the rest of the output.

tty_read MODULE

A user process that is ready to handle terminal input calls one of several entries in tty_read, which copies whatever input is available from the specified channel into a buffer supplied by the caller. If there is no available input from the channel, the caller normally goes blocked; a wakeup is sent when input arrives from the FNP.

tty_index MODULE

The tty_index module contains a variety of entries concerned with control of a channel or the data bases associated with it. The tty_index, tty_attach, tty_event, tty_detach, and tty_new_proc entries deal with the associations between communications channels and processes; the tty_order and tty_abort entries are used to send control information about the channel to the FNP and to modify the treatment by Multics Communication System of input and output data.

Multiplexing Interfaces

Two call-switching modules, channel_manager and priv_channel_manager, are used to pass calls and interrupts from one level of multiplexing to the next. They are described in more detail in Section 3.

The standard FNP multiplexer module, fnp_multiplexer, is called by channel_manager to send output and control information to an FNP. Its task is to format an FNP mailbox and pass it on to dn355.

The interrupt handler for nonmultiplexed channels is tty_interrupt. This is the module that sends wakeups to user processes when input arrives or output finishes.

SOME STANDARD SEQUENCES

Channel Initialization

As each FNP or concentrator channel is initialized, the answering service takes control of each subchannel of the FNP or concentrator as defined in the channel definition table (CDT) by calling tty_attach for each one. After a channel is attached, the answering service issues a "listen" request by calling tty_order; tty_order forwards the request to fnp_multiplexer, which encodes it into a mailbox to send to the FNP. Once the FNP has received a "listen" request for the channel, it is prepared to accept dialups from the channel.

Dialup

When a connection is established between the FNP and a communications channel, the FNP sends a mailbox to the CS with the operation code "accept new terminal" (see Appendix A for a description of mailbox operation codes).

The dn355 module sends a mailbox back to the FNP saying "terminal accepted," and forwards the interrupt to tty_interrupt, which wakes up the answering service to inform it of the dialup. If a user then logs in from the newly-dialed-up terminal, a process is created, and the answering service "lends" the channel to this process by calling tty_new_proc, thus establishing the new process as the "user" of the channel.

Output

When the user of the channel calls tty_write, his data (or as much of it as can be handled at once) is copied first into the internal buffers of tty_write (see Section 5); tty_write then performs the necessary conversions and translations, copies the data into tty_buf (either starting an output chain for the channel, or appending to an already existing one), and calls channel_manager$write. The call is eventually forwarded to dn355, which sends a mailbox to the FNP telling it that there is output for the channel, and also telling how much and where in tty_buf it is; the FNP then copies the output from tty_buf into FNP memory in preparation for transmitting it to the channel. Once the FNP has copied the output, dn355 frees the buffers composing the output chain in tty_buf.

Input

When a "break" character (i.e., an end-of-message character, typically a newline) is input, the FNP sends a mailbox informing the CS that input of a certain length has been received over the channel. If there is room for it, dn355 replies with a mailbox instructing the FNP to copy the input into the circular buffer in the header of tty_buf (see Section 2); once the FNP I/O has completed, dn355 allocates an input chain in tty_buf, copies the data into this chain from the circular buffer and sends an "accept input" interrupt to tty_interrupt. If the process using the channel has unsuccessfully attempted to obtain input from the channel (as indicated by a flag in the wired terminal control block (WTCB)), a wakeup is sent to inform it that the input has arrived.

When the user process calls tty_read (either because it received the wakeup, or simply because it was ready to receive input), the data is copied into the internal buffers of tty_read, translated and converted appropriately, and copied into the buffer supplied by the caller. Once it has processed the input, tty_read frees the input chain buffers in tty_buf.

Quit

When a user presses the ATTENTION or INTERRUPT key at a terminal, a "line break" condition is generated, which is recognized by the FNP and reflected back to the CS. If the channel is in "hndlquit" mode (which it usually is), dn355 discards any pending output currently in tty_buf intended for the physical channel. The interrupt is then forwarded through channel_manager to tty_interrupt, which discards any pending input and output from or to the logical channel. If quits have been "enabled" through the use of the quit_enable control operation (as they almost always are), tty_interrupt calls

pxss$ips_wakeup_int, which causes the "quit" condition to be signalled in the user process.

Hangup

A hangup (disconnection of the channel) can be initiated by the CS (as a result of a "hangup" control request sent through tty_order), in which case the FNP takes action to disconnect the channel; or the FNP may detect that the channel has disconnected, as a result of either deliberate action by the user of the terminal or failure of the communications equipment. In either case, once the FNP is satisfied that the connection has been broken, it sends a mailbox with the operation code "terminal disconnected." On receiving this mailbox, dn355 forwards the interrupt to tty_interrupt, which sends a wakeup to the "owning" process (generally the answering service) and frees any input and output chains associated with the channel. After receiving the hangup wakeup, the answering service usually issues another "listen" request so that the channel can be dialed up again.

If the channel that hung up is a multiplexed channel, the interrupt handler for that channel sends a "crash" interrupt to tty_interrupt for each currently active subchannel. These interrupts are handled exactly like hangup interrupts, except that tty_interrupt does not wake up the owning process, since the subchannels cannot be listened to again until the multiplexer is reconnected. The interrupt handler for the major channel does wake up the answering service, which then takes appropriate action to reinitialize the major channel and its subchannels, as described in Sections 3 and 7.

SECTION 2

HARDCORE DATA BASES

DATA BASES USED FOR COMMUNICATION WITH FNP

Two wired segments are used by the Multics hardcore for communication with, and storing information about, the various configured FNPs. These segments are dn355_data, which describes the FNP configuration and current status of each FNP, and dn355_mailbox, which is used for direct communication with the FNP as described in Section 4.

dn355_data

The format of dn355_data is described in the include file dn355_data.incl.pl1. The first part of the segment contains general configuration information; the remainder consists of one block of FNP-specific information for each configured FNP. This block (called fnp_info) is the multiplexer database for the FNP; a pointer to the corresponding block is kept in each FNP's LCT entry (see the discussion of the LCT later in this section) and passed to the various entries in fnp_multiplexer. Included in the fnp_info block is a pointer to the FNP's mailbox area and a pointer to an array of physical channel blocks (PCBs) containing information relevant to the individual subchannels of the FNP.

dn355_mailbox

This segment contains a mailbox area for each configured FNP. 300(8) words are reserved for each mailbox area.

Each mailbox area consists of an 8-word header followed by twelve submailboxes. The submailboxes are used for individual communications between the FNP and the CS. The first eight submailboxes are eight words each, and are used for transactions originating in the CS. The remaining four submailboxes are 28 words each, and are used for transactions originating in the FNP. See Section 4 for further information on the use of submailboxes.

The formats of the mailbox header and the submailboxes are described in the include file dn355_mailbox.incl.pl1. The mailbox header includes the following items: the peripheral control word (PCW) used to send interrupts to the FNP over the DIA; an array of flags indicating which CS-controlled submailboxes are currently in use; and a "terminate interrupt multiplex word" (timw) used by the FNP to indicate that it has processed a particular submailbox. Each bit that is on in the timw corresponds to a submailbox used by the FNP since the last time dn355 examined it. The mailbox header also contains two words of "crash data" that are filled in by the fault handler of the FNP when the FNP crashes, giving the type of fault that caused the crash, the FNP instruction counter at the time of the fault, and, if applicable, encoded information enabling dn355 to find an appropriate error message in dn355_messages (see Section 8).

Each submailbox contains an I/O command and an operation code (opcode). The I/O command indicates which of four classes of operations the submailbox specifies; the opcode indicates the specific operation. The four possible I/O command values are:

1 = read control data (rcd):
control information being passed from the FNP to the CS.

2 = read text (rtx):
terminal input being passed from the FNP to the CS.

3 = write control data (wcd):
control information being passed from the CS to the FNP.

4 = write text (wtx):
terminal output being passed from the CS to the FNP.

The opcodes are summarized in Appendix A. Opcodes and I/O commands are defined in mailbox_ops.incl.pl1.

A submailbox may also contain additional information describing the operation in detail. The meaning of this additional information varies according to the opcode. Each submailbox includes in its low-order 18 bits a checksum which is the sum of all the 9-bit bytes in the remainder of the submailbox.

tty_buf

The wired segment tty_buf is used by all portions of the hardcore communications system. It contains information of general interest to Multics Communication System, the control blocks associated with each communications channel, and the pool of free space in which input and output buffers are allocated.

Its size can be set by means of a PARM TTYB card in the configuration deck, e.g.:

PARM TTYB 8192.

The default size is 5120 words (5K).

The layout of tty_buf is as follows: the first part is a header, containing information about the current state of the communications system and a variety of metering information; then comes a circular buffer into which terminal input is written by the FNP; following this is the logical channel table (LCT), allocated during initialization. The remainder of the segment is free space, in which are allocated data buffers, various control blocks and delay queues.

tty_buf Header

The format of the tty_buf header is described in the include file tty_buf.incl.pl1. It contains control words giving the origin and length of the free space pool, the size of the circular buffer, and the address of the LCT. It also is used to store a variety of metering information relating to Multics Communication System, which is copied out as needed by the tty_meters command. The first word of the tty_buf header is a lock used by tty_space_man to ensure that no two processors attempt to update the free space pool at the same time.

Circular Buffer

The circular buffer (also called the circular queue) is a buffer in which long input messages are stored directly by the FNP. (Short messages are sent in submailboxes; see Section 4 for more information.) Its size may be set by a PARM TTYQ card in the configuration deck, e.g.:

PARM TTYQ 1024.

The default size is 256 words. The circular buffer cannot be smaller than 256 words.

Logical Channel Table (LCT)

The logical channel table (LCT) consists of a header and an array of LCT entries (LCTEs). There is an LCTE for every channel (at each level of multiplexing) defined in the CDT. The array is allocated in tty_buf at system initialization time; spare entries may be allocated in case the number of configured channels is to be increased while the system is running. The

number of such spare entries is determined by the spare_channel_count keyword in the CMF; the default is 10. The format of the LCT is described in the include file lct.incl.pl1.

The LCT header contains the size of the array of LCTEs, a pointer to the logical channel name table (LCNT), which is discussed later in this section, and a global lock for the delay queues (see the discussion of locking and queuing in Section 6).

The position of a channel's LCTE in the LCTE array is that channel's device index (devx), which is the number used throughout ring zero to identify the channel. The contents of an LCTE include: the "multiplexer type" of the channel (e.g., FNP, "tty" or nonmultiplexed channel, etc.); a pointer to the database associated with the channel; the devx of the channel's parent multiplexer or "major channel"; the subchannel of the major channel that this channel represents; the devx of the physical channel (i.e., subchannel of an FNP) of which this channel is a descendant (which is the same as the channel's own devx if the LCTE is that of a physical FNP channel); a lock used to ensure that the LCTE is never modified by more than one process at a time; and various flags. An entry-in-use flag is used to indicate whether the LCTE is currently valid; this flag is turned on when the channel is initialized, and turned off when it is terminated. (See the discussions of multiplexing in Section 3 and initialization in Section 7 for more details.)

The information in the LCTE is used by channel_manager (described in Section 3) to determine what multiplexer module to forward calls to, and to enable it to identify the multiplexer channel and subchannel to the called entry.

Dynamic Portion of tty_buf

The remainder of tty_buf is allocatable space, which may be used for any of the following purposes:

- free block
- data buffer (input or output)
- per-channel control block
- output pseudo-DCW list
- delay queue entry

FREE BLOCK

A free block is a contiguous block of any even number of words that is available for allocation by tty_space_man (see the discussion of space management in Section 6). It has a header that contains its size in words and the offset in tty_buf of the next free block. Free blocks are chained together in order of

increasing address; a word in the tty_buf header contains the offset of the first free block in the chain. Adjacent free blocks are combined into larger blocks as they become free.

DATA BUFFER

A data buffer contains either input characters that have come from a channel or output characters that are to be sent to a channel. The input or output for a particular channel are organized into chains of such buffers (called input chains and output chains). Each buffer in such a chain is a block whose size is a multiple of 16 words, up to a limit of 128 words. The first word of a buffer is a control word containing the offset of the next buffer in the chain, a size code indicating the size of the buffer, the tally of valid characters actually present in the buffer, and some flags. The remainder of the buffer contains data charaters. The size code has a value between 0 and 7 inclusive, and is one less than the size of the buffer in multiples of 16 words; in other words, a code of 0 indicates a size of 16 words, a code of 1 indicates a size of 32 words, and a code of 7 indicates a size of 128 words. The format of a data buffer is described in the include file tty_buffer_block.incl.pl1.

PER-CHANNEL CONTROL BLOCKS

Per-channel control blocks are allocated as needed in tty_buf when each channel is initialized. The format of such a control block depends on the type of the channel. Two standard formats are described here: the wired terminal control block and the physical channel block.

Wired Terminal Control Block (WTCB)

The wired terminal control block (WTCB) is the primary database for a nonmultiplexed channel; the database pointer in the LCTE of such a channel points to its WTCB. The WTCB is used by tty_read, tty_write, tty_index, and tty_interrupt. Its format is described in the include file wtcb.incl.pl1.

The WTCB contains all information about the current state of the channel that is ever needed or modified by tty_interrupt, which cannot reference unwired data. This information includes the identification of the "owning" and "user" processes for the channel, and the event channels used for waking up these processes; the baud rate and line type of the channel, which are set at dialup time; the offsets of any currently-active input and output chains; and various flags. The WTCB also contains a pointer to the unwired terminal control block (TCB), described later in this section.

Physical Channel Block (PCB)

The physical channel block (PCB) contains information specific to a physical subchannel of an FNP. It is used by fnp_multiplexer and dn355. The format of the PCB is described in the include file pcb.incl.pl1.

The PCBs for the subchannels of an FNP are allocated in a contiguous array when the FNP is initialized; the fnp_info block for each FNP contains a pointer to the beginning of the PCB array for that FNP. The subchannel number of each physical channel is the index into the PCB array of that channel's PCB.

The PCB contains flags describing the current state of the physical channel, and pointers to any output chain that has been passed to fnp_multiplexer but not yet sent over the DIA.

PSEUDO-DCWS

An output "pseudo-DCW" list is used to send an FNP the addresses and tallies of the individual buffers in an output chain. An output chain is described by an array of up to 16 pseudo-DCWs in a single allocated block. Each pseudo-DCW occupies one word, containing the absolute address of an output buffer and the number of characters in the buffer.

DELAY QUEUE ENTRIES

The delay queue for a channel contains entries describing interrupt events for that channel that could not be processed because the channel's LCTE was locked at the time of the interrupt. These queue entries are processed when it is time to unlock the LCTE (see Section 6 for details). A delay queue entry contains the offset in tty_buf of the next entry in the channel's queue (if any) and the interrupt type and data that were passed to channel_manager$interrupt (see Section 3). If a delay queue exists for a channel, the offsets of its first and last entries appear in the channel's LCTE.

tty_area

The unwired ring-zero segment tty_area is used for control blocks that are not needed by any wired programs. It is managed by means of the PL/I area mechanism.

Two types of databases are allocated in tty_area: the logical channel name table (LCNT) and (unwired) terminal control blocks (TCBs).

Logical Channel Name Table (LCNT)

The LCNT is an array of channel names. Each name in the LCNT occupies the same relative position as the named channel's LCTE in the LCT. Thus the LCNT can be used to derive a channel's name from its devx or vice versa. The format of channel names is described in MPM Communications Input/Output, Order No. CC92.

Terminal Control Block (TCB)

There is a TCB for each initialized nonmultiplexed channel. It contains information used at call time for the conversion and translation of input and output data as described in Section 5. The format of the TCB is described in the include file tcb.incl.pl1. The TCB for each channel is found by following a pointer in the channel's WTCB (see above).

The TCB includes pointers to the various conversion and translation tables kept in tty_tables (discussed below). For each type of table, two relative pointers are kept in the TCB: a current table pointer and a default table pointer. The default table pointer is used to reset the current table to the default for the channel's terminal type. If the default table pointer is -1, the current table is the default table; otherwise, the current table has been set explicitly, and the default table is made the current table if an order is made to set the table to its default value.

tty_tables

The unwired segment tty_tables contains all the tables to be used in converting terminal input and output between the form stored internally and the form in which it is received from or sent to the terminal. These tables are supplied by user-ring programs by means of the set_input_translation, set_output_translation, set_input_conversion, set_output_conversion, set_special, and set_delay orders.

The tty_tables segment is managed by a utility program named tty_tables_mgr, which is called by tty_index. The segment consists of a header and an area in which the tables are allocated. Each table in the area is preceded by a descriptor that identifies the table type and links tables of that type in a list. Six table types are supported:

1. input translation

2. output translation

3. input conversion

4. output conversion

5. special

6. delay

The tty_tables header contains a relative pointer to the first table of each type. The format of the tty_tables header and the table descriptor are described in the include file tty_tables.incl.pl1; the formats of the tables themselves are described in the include file tty_convert.incl.pl1; the tables are discussed in more detail in the description of the tty_ I/O module in MPM Communications Input/Output, Order No. CC92.

The tables are shared as necessary, i.e., if two channels are using identical tables of any type, only one copy of the table is kept in tty_tables. Accordingly, a reference count is kept in the table descriptor to indicate how many TCBs contain pointers to the table; a table is not freed until the reference count goes to zero.

SECTION 3

MULTIPLEXING

MULTIPLEXED CHANNELS

The device associated with a physical FNP channel may be some kind of concentrator that controls multiple terminals (examples of such concentrators include the Honeywell VIP 7700 series and the IBM Model 3270 series). Multics Communication System is capable of treating each such terminal as a separate logical channel. In this case, the channel occupied by the concentrator is called a multiplexed channel, and the concentrator is referred to as a multiplexer; the logical channels associated with the individual terminals are called subchannels of the multiplexer. The multiplexed channel and its subchannels must all be defined in the CDT. A subchannel of a multiplexer might itself be multiplexed. Such multiplexing can be carried to any number of levels.

Since an FNP controls many channels, but communicates with the central system over a single channel (the DIA), the FNP may be regarded as a multiplexer. Frequent reference is made in this manual to a channel's multiplexer or "parent multiplexer" (i.e., the multiplexer of which it is a subchannel); when the channel referred to is a physical FNP channel, the parent multiplexer is the FNP. A channel's parent multiplexer is also sometimes called its "major channel."

Basic Design

The basic design for ring 0 demultiplexers centers around a database called the logical channel table (LCT) and a class of programs called multiplexer modules.

The LCT contains one entry for every channel and subchannel managed by Multics Communication System. An LCT entry is identified by a device index (devx). LCT entries correspond to various levels of multiplexing. Starting at the bottom level, there is one LCT entry for each FNP. At the next level, there is one LCT entry for each physical channel on each FNP. If one of

these physical channels is multiplexed, then there is one LCT entry for each subchannel. A subchannel itself can be multiplexed, in which case there is yet another level of subchannels.

A separate multiplexer module, or group of modules, is required for each type of multiplexed device. Each such module must provide a standard set of interfaces that are invoked through a call switch named channel_manager. Some of these interfaces are invoked in response to user calls while others are invoked in response to interrupts. Call-side operations are routed through a series of transitions from subchannel to major channel whereas interrupts follow the reverse path. At each transition, channel_manager is invoked to select the appropriate multiplexer module as determined by the channel type of the target channel. In the interrupt case, the multiplexer module is also referred to as an interrupt handler.

Taken together, the LCT and the multiplexer modules yield a systematic approach to handling arbitrary levels of multiplexing. Each call-side operation propagates downward through one or more levels of multiplexing until reaching the FNP multiplexer. Each interrupt-side operation propagates upward through one or more levels of demultiplexing until reaching a nonmultiplexed logical channel.

MULTIPLEXER TYPES

Each logical communications channel has associated with it a channel type or "multiplexer type", which is specified in its entry in the channel definition table (CDT) (see the Multics Administrators' Manual -- Communications, Order No CC75, for a description of the CDT). Honeywell supplies multiplexer modules to support several types of multiplexers, and others can be added by sites to suit their individual needs (which requires modifications and additions to the supervisor). The name of a multiplexer module is associated with the multiplexer type in a transfer vector module, cmtv, which is described later in this section. The following types are reserved for system use:

tty	(nonmultiplexed channel)
mcs	(DN6600 or DN6670 FNP using Multics Communication System protocol)
ibm3270	(IBM Model 3270 controller)
vip7760	(Honeywell VIP7760 and 7700 series)

In addition, the types user1, user2, user3, user4, and user5 are available for site-supplied multiplexer modules.

The remainder of this section describes the general structure of multiplexing in ring 0, including the interfaces to be used and the conventions to be followed in writing additional multiplexer modules.

CALLS AND INTERRUPTS

The ring 0 portion of Multics Communication System is driven both by calls from outer rings (for output and control operations) and by interrupts from the FNP (for input and status reporting). Calls enter ring 0 at the "leaf node" level, i.e., at the level of nonmultiplexed terminal channels; entries that are called directly through gates into ring 0 are all in the modules tty_read, tty_write, and tty_index (with the exception of a few privileged operations performed directly on multiplexed channels, described later in this section). If the operation requested by the caller requires the involvement of the major channel, the called module calls one of the standard entries in channel_manager, which forwards the call to the corresponding entry in the appropriate multiplexer module. (These entries are described later in this section.) The multiplexer module may, in turn, pass the call on to its major channel (again through channel_manager), and so on, until eventually the call reaches the multiplexer for the physical channel; for channels of a standard FNP (DN6600 or DN6670), this module is fnp_multiplexer.

Interrupts sent by the FNP over the DIA are handled by dn355$interrupt. This entry figures out which subchannel of the FNP (if any) the interrupt is intended for, and calls channel_manager$interrupt in order to pass the interrupt to the appropriate handler for the subchannel. The arguments passed include the interrupt type and any associated data. If the FNP subchannel is multiplexed, and the interrupt is for one of its subchannels, the interrupt handler calls channel_manager$interrupt again, and thus the interrupt is passed along until it reaches tty_interrupt, the interrupt handler for nonmultiplexed channels. The tty_interrupt module takes the action appropriate to the interrupt type, which often includes sending a wakeup to the process that has the channel attached.

channel_manager AND priv_channel_manager

The call-switching function of channel_manager is actually divided between two modules, channel_manager itself and priv_channel_manager. The latter is used to forward initialization and special-purpose control calls, most of which are applied to the multiplexer channel itself rather than one of its subchannels.

Entries to channel_manager

In any given multiplexer module, there is an entry corresponding to, and having the same name as, each entry in channel_manager. The arguments to channel_manager are passed through to the multiplexer module, with the exception of the devx, which channel_manager translates into a database pointer and a subchannel number. The database pointer points to a database whose format and use depends on the particular multiplexer type; each multiplexer maintains one such database for each major channel of its type. This database is constructed by the init_multiplexer entry of the multiplexer (see below), which returns a pointer to the database. This pointer is kept in the major channel's LCT entry.

The following is a summary of the entries to channel_manager, including a brief description of the purpose of, and the arguments to, each entry. These entries are all declared in the include file channel_manager_dcls.incl.pl1. Unless otherwise noted, for calls to multiplexer modules, the devx in each entry is replaced by two input arguments: the database pointer and the subchannel number.

Entry: channel_manager$read

This entry is called to obtain any input for the specified subchannel that is being held by the multiplexer. Such input is passed to the caller in chained buffers allocated in tty_buf, as described in Section 2; the number of characters in each buffer is provided in the tally field of the buffer. In general, a multiplexer module need not call channel_manager$read unless an input_available interrupt (see below) has been received.

Usage

```
declare channel_manager$read (fixed bin, ptr, bit(1)
     aligned, fixed bin(35));

call channel_manager$read (devx, chain_ptr, more_input_flag,
     code);
```

where:

chain_ptr (Output)

is a pointer to the first buffer in the chain. If no input is available, chain_ptr is set to null.

more_input_flag (Output)

is set to "1"b if there is additional input for the subchannel that has not returned in the input chain.

code (Output)

is a standard system status code.

Entry: channel_manager$write

This entry is called to send output to the specified subchannel. The output is passed in a buffer chain like the one described for the read entry above. The multiplexer entry may perform further processing on the output data if necessary (for instance, to put it into a format recognized by the multiplexing device). The multiplexer entry may, for whatever reason, accept all, part, or none of the output; this is reflected by the returned value of the chain_ptr.

NOTE: A multiplexer module should never call channel_manager$write for a particular multiplexer channel unless a send_output interrupt (see below) has been received for that channel.

Usage

```
declare channel_manager$write (fixed bin, ptr, fixed
     bin(35));

call channel_manager$write (devx, chain_ptr, code);
```

where:

chain_ptr (Input/Output)

is a pointer to the first buffer in the output chain. If all the output in the chain is accepted by the write entry of the multiplexer, chain_ptr is set to null; otherwise it is set to the address of the first buffer of the remainder of the chain. In the latter case, the chain is not guaranteed to occupy the same buffers after the call as before; it is the responsibility of the caller to examine the chain pointed to by the new chain_ptr.

code (Output)

is a standard status code. It may be error_table_$noalloc to indicate that insufficient space was available in tty_buf to process the output.

Entry: channel_manager$control

This entry is called to perform a control operation on the specified subchannel. The set of operations supported may vary depending on the multiplexer type, but must include the following:

listen
hangup
write_status
abort
wru

Usage

```
declare channel_manager$control (fixed bin, char(*), ptr,
     fixed bin(35));

call channel_manager$control (devx, control_type, info_ptr,
     code);
```

where:

control_type (Input)

is the name of the operation to be performed.

info_ptr (Input)

is a pointer to any additional data required to specify the operation. The format of this data depends on the type of operation. If no data is supplied, info_ptr should be null.

code (Output)

is a standard system status code. It may be error_table_$undefined_order_request to indicate that the specified operation is not supported for this multiplexer type.

Entry: channel_manager$check_modes

This entry is called to ascertain which, if any, of the specified modes are recognized by the multiplexer and whether the specified set of modes is valid. The caller should provide an entry in the modes change list structure described below for each mode it intends to set, to find out if the multiplexer needs to set the mode as well.

Usage

```
declare channel_manager$check_modes (fixed bin, ptr, fixed
     bin(35));

call channel_manager$check_modes (devx, mclp, code);
```

where:

mclp (Input)

is a pointer to the structure described under "Notes," below.

code (Output)

is a standard system status code. It may be error_table_$bad_mode to indicate that one or more of the specified modes is recognized by the multiplexer, but cannot be set for the specified subchannel.

Notes

The mclp argument must point to the structure described below, which is defined in the include file mcs_modes_change_list.incl.pl1:

```
declare 1 mcl aligned based,
          2 version fixed bin,
          2 n_entries fixed bin,
          2 line_len fixed bin,
          2 page_len fixed bin,
          2 flags
            3 init bit(1) unaligned,
            3 ll_error bit(1) unaligned,
            3 pl_error bit(1) unaligned,
            3 mbz bit(33) unaligned,
          2 entries (36) like mcle;
```

where:

version (Input)

is the version number of the structure; it must be 1.

n_entries (Input)

is the number of entries (see below) that are used.

line_len (Input)

if the new setting of the line length ("ll") mode, or -1 if the line length is not being changed.

page_len (Input)

is the new setting of the page length ("pl") mode, or -1 if the page length is not being changed.

init (Input)

is "1"b if the "init" mode was specified, indicating that all unspecified modes are to be turned off. (This flag may be ignored by the check_modes entry, but is used by the set_modes entry described below.)

ll_error (Output)

is set to "1"b if the supplied line length cannot be set.

pl_error (Output)

is set to "1"b if the supplied page length cannot be set.

entries

are entries for the individual modes (one for each mode), in the format described by the following structure:

```
dcl 1 mcle aligned based,
    2 mode_name char(16) unaligned,
    2 flags
      3 mode_switch bit(1) unaligned,
      3 force bit(1) unaligned,
      3 mpx_mode bit(1) unaligned,
      3 error bit(1) unaligned,
      3 mbz bit(32) unaligned;
```

where:

mode_name (Input)

is the name of the mode.

mode_switch (Input)

is "1"b if the mode is to be turned on, or "0"b if it is to be turned off.

force (Input)

is "1"b if no error indication is to be returned for this mode.

mpx_mode (Output)

is set to "1"b if the mode is recognized as valid by the multiplexer.

error (Output)

is set to "1"b if the mode is recognized as invalid by the multiplexer and force (above) is "0"b. If this flag is turned on in any entry, a code of error_table_$bad_mode is returned.

Entry: channel_manager$set_modes

This entry is called to change the setting of the specified modes for the specified subchannel. This entry should not be called unless the check_modes entry (see above) has returned a code of 0 for the specified modes and subchannel.

Usage

```
declare channel_manager$set_modes (fixed bin, ptr, fixed
     bin(35));

call channel_manager$set_modes (devx, mclp, code);
```

where:

mclp (Input)

is as described for the check_modes entry, above. The mpx_mode flag in each entry in the structure is now to be taken as an input argument; if it is "1"b, the mode is to be set by the multiplexer.

code (Output)

is a standard system status code.

Entry: channel_manager$get_modes

This entry is called to find out the current settings of modes known to the multiplexer for the specified subchannel.

Usage

```
declare channel_manager$get_modes (fixed bin, char(*), fixed
     bin(35));

call channel_manager$get_modes (devx, modes, code);
```

where:

modes (Output)

is set to the list of modes known to the multiplexer, separated by commas. Modes that are currently off for the specified subchannel are preceded by a ^ character.

code (Output)

is a standard system status code. It may be error_table_$smallarg to indicate that the modes argument (above) was not long enough to hold the entire list of modes (in which case the list is returned, but truncated).

Entry: channel_manager$interrupt

This entry is called to forward interrupts from a multiplexer's interrupt handler to the interrupt handler for one of its subchannels. The database pointer provided by channel_manager points to the database of the logical channel receiving the interrupt; accordingly, no subchannel number is passed. The interrupt handler may, on the basis of information provided with the interrupt, forward the interrupt to one of its subchannels.

Usage

```
declare channel_manager$interrupt (fixed bin, fixed bin,
     bit(72) aligned));

call channel_manager$interrupt (devx, int_type, int_data);
```

where:

int_type (Input)

is the type of interrupt. Interrupt types are summarized below.

int_data (Input)

is any additional data required to describe the interrupt. The data associated with each interrupt type is summarized below.

INTERRUPT TYPES AND ASSOCIATED DATA

The interrupt types and the structures of the associated data are defined in mcs_interrupt_info.incl.pl1. The summaries below are intended to supplement, not replace, the information by the include file.

Type:
dialup

Associated Data:
line type, baud rate, maximum output buffer size.

Purpose:
report that the channel has dialed up and is available for I/O.

Type:
hangup

Associated Data:
none

Purpose:
report the logical or physical disconnection of the channel. This information is ultimately passed on to the answering service, which will listen for further dialups on the channel.

Type:
crash

Associated Data:
none

Purpose:
same as hangup, except that the answering service is not to listen for further dialups. This interrupt type is used when the channel's parent has crashed or hung up.

Type:
send_output

Associated Data:
none

Purpose:
inform the handler that the channel is ready to accept output.

Type:
input_available

Associated Data:
none

Purpose:
inform the handler that the multiplexer has input for the channel. This input can be picked up by a call to channel_manager$read. See "Notes" below for more information.

Type:
accept_input

Associated Data:
chain pointers, number of characters, flags.

Purpose:
pass input on to the subchannel. The input chain is in buffers in tty_buf. After receiving this interrupt, the handler is responsible for further disposition of the input.

Type:
    input_rejected

Associated Data:
    none.

Purpose:
    inform the handler that input for this channel has been rejected by the FNP interrupt handler because of space problems. Tne handler should take whatever action is appropriate (such as sending wakeups, or passing the interrupt on to any of its subchannels that have input) to free any buffer space for which it is responsible.

Type:
    quit

Associated Data:
    none

Purpose:
    report that a quit (line break, interrupt) signal has been received from the subchannel.

Type:
    line_status

Associated Data:
    status information from the FNP

Purpose:
    report status reported by the FNP control tables that run the channel's line type.

Type:
    dial_status

Associated Data:
    status of a dialout operation.

Purpose:
    report the result of a dialout operation to a channel equipped with an automatic calling unit (ACU). This interrupt is unlikely to be meaningful for any channel other than a physical subchannel of an FNP.

Type:
    wru_timeout

Associated Data:
none

Purpose:
report that a "who-are-you" operation sent to the channel received no response.

Type:
space_available

Associated Data:
none

Purpose:
report that space that has been requested for an output operation is (or may be) now available; any pending output should be retried.

Notes

A multiplexer that needs to perform extensive transformations on its input data should send input_available interrupts to its subchannels, so that the transformations can be performed when a read call is made rather than at interrupt time. If major transformations are not required, the accept_input interrupt mechanism is likely to be more efficient.

Entries in priv_channel_manager

The entries in priv_channel_manager are called to perform privileged operations on communications channels. Most of these calls apply to multiplexer channels themselves rather than to nonmultiplexed subchannels. The most commonly used of these calls originate in the initializer process. Unless otherwise specified, the privileged entries can be reached from outer rings only through the hphcs_ gate.

The entries to priv_channel_manager are summarized below. Unless otherwise specified, the devx argument is replaced by a database pointer in forwarding the call to the corresponding entry in the multiplexer module, and other arguments are unchanged.

Entry: priv_channel_manager$init_multiplexer

This entry is called by the answering service to cause the multiplexer module to initialize its databases. The answering service makes one such call for every multiplexed channel of an FNP when the FNP is loaded, and for each subchannel of such a

multiplexed channel that is also multiplexed. Each multiplexer must be initialized before it can be loaded or any of its subchannels can be used.

Usage

```
declare priv_channel_manager$init_multiplexer (fixed bin,
     fixed bin, ptr, fixed bin(35));

call priv_channel_manager$init_multiplexer (devx, chan_type,
     mux_init_info_ptr, code);
```

where:

devx (Input)

is the device index of the multiplexer channel.

chan_type (Input)

is the channel's multiplexer type.

mux_init_info_ptr (Input)

is a pointer to a structure identifying all the subchannels of the multiplexer. This structure contains the name of each subchannel; priv_channel_manager fills in the devx of each subchannel before passing the structure on to the multiplexer module. The format of the structure is described by the include file mux_init_info.incl.pl1.

code (Output)

is a standard system status code.

The call is forwarded to the init_multiplexer entry or the multiplexer module as follows:

```
declare <name>$init_multiplexer (fixed bin, ptr, ptr, fixed
     bin(35));

call <name>$init_multiplexer (devx, mux_init_info_ptr,
     data_base_ptr, code);
```

where:

devx (Input)

is as above.

mux_init_info_ptr (Input)

is as above.

data_base_ptr (Output)

is a pointer to the multiplexer database for this major channel. It will be passed back to the multiplexer module in subsequent calls in order to identify the major channel. The init_multiplexer entry is expected to allocate the database in tty_buf (see Section 6 for a discussion of space management).

code (Output)

is as above.

Entry: priv_channel_manager$terminate_multiplexer

This entry is called after a multiplexer has been shut down or crashed, and after all its subchannels have been terminated. Its primary purpose is to free the multiplexer database and perform any other necessary cleaning up. In general, if a multiplexer crashes or hangs up it will be terminated and reinitialized automatically by the answering service.

Usage

```
declare priv_channel_manager$terminate_multiplexer (fixed
     bin, fixed bin(35));

call priv_channel_manager$terminate_multiplexer (devx,
     code);
```

where arguments are as above.

Entry: priv_channel_manager$start

This entry is called to make a loaded and initialized multiplexer active, i.e., to enable listening on its subchannels. It is called either automatically after the multiplexer is loaded, or in response to an operator command of start_mpx.

Usage

```
     declare priv_channel_manager$start (fixed bin, fixed
          bin(35));

     call priv_channel_manager$start (devx, code);
```

where arguments are as above.

Entry: priv_channel_manager$stop

This entry is called to prevent dialups from coming in from the subchannels of a multiplexer. Currently active subchannels are not affected. It is invoked as a result of the operator command stop_mpx.

Usage

```
     declare priv_channel_manager$stop (fixed bin, fixed
          bin(35));

     call priv_channel_manager$stop (devx, code);
```

where arguments are as above.

Entry: priv_channel_manager$shutdown

This entry is called to force disconnection of all subchannels of a multiplexer. It is called whenever the multiplexer crashes or hangs up, and whenever its parent multiplexer is shut down.

Usage

```
     declare priv_channel_manager$shutdown (fixed bin, fixed
          bin(35));

     call priv_channel_manager$shutdown (devx, code);
```

where arguments are as above.

Entry: priv_channel_manager$priv_control

This entry is called to perform a control operation on the multiplexer channel itself (as opposed to one of its subchannels). The particular operations supported depend on the multiplexer type. This entry is accessible only through the gate phcs_.

Usage

```
declare priv_channel_manager$priv_control (char(*), char(*),
     ptr, fixed bin(35));

call priv_channel_manager$priv_control (chan_name,
     control_type, info_ptr, code);
```

where:

chan_name (Input)

is the name of the multiplexer channel. It is replaced by the database pointer in forwarding the call to the multiplexer module.

control_type (Input)

is the name of the control operation.

info_ptr (Input)

is a pointer to any additional data associated with the control operation. If there is no such data, info_ptr should be null.

code (Output)

is a standard system status code. It may be error_table_$undefined_order_request to indicate that the specified control_type is not supported.

Entry: priv_channel_manager$hpriv_control

This entry is exactly like the priv_control entry described above, except that it is accessible only through the gate hphcs_.

Usage

```
declare priv_channel_manager$hpriv_control (char(*),
     char(*), ptr, fixed bin(35));

call priv_channel_manager$hpriv_control (chan_name,
     control_type, info_ptr, code);
```

where arguments are the same as in priv_channel_manager$priv_control.

Entry: priv_channel_manager$init_channel

This entry is called to initialize a nonmultiplexed channel. A site-supplied multiplexer module need not include a corresponding entry.

Usage

```
declare priv_channel_manager$init_channel (fixed bin, ptr,
     fixed bin(35));

call priv_channel_manager$init_channel (devx, info_ptr,
     code);
```

where:

devx (Input)

is the device index of the channel.

info_ptr (Input)

is a pointer to additional data needed to initialize the channel (currently not used).

code (Output)

is as above.

Entry: priv_channel_manager$terminate_channel

This entry is called to terminate a nonmultiplexed channel. A site-supplied multiplexer module need not include a corresponding entry.

Usage

```
     declare priv_channel_manager$terminate_channel (fixed bin,
          fixed bin(35));

     call priv_channel_manager$terminate_channel (devx, code);
```

where arguments are as above.

Entry: priv_channel_manager$get_devx

This entry is called by an outer ring procedure to obtain the devx of a channel whose name is known. This call is not forwarded by priv_channel_manager.

Usage

```
     declare priv_channel_manager$get_devx (char(*), fixed bin,
          fixed bin(35));

     call priv_channel_manager$get_devx (chan_name, devx, code);
```

where:

chan_name (Input)

is the name of the channel whose devx is to be returned.

devx (Output)

is the devx of the channel.

code (Output)

is as above.

MULTIPLEXER TRANSFER VECTOR -- cmtv

A transfer vector module written in ALM, named cmtv, is used by channel_manager and priv_channel_manager to forward calls to the appropriate multiplexer and interrupt handler entries. The source program cmtv.alm includes definitions of assembler macros used for defining the entries to be called for any specified multiplexer type. A macro statement must be included for each multiplexer type supported by a given site. The entries

specified for a multiplexer type may be all in one module or distributed among several modules. The source of cmtv contains comments describing the format of the required macro statements; the definitions of existing multiplexer types may be taken as models.

PROGRAMMING CONSIDERATIONS -- WIRED CODE

Because the interrupt handler portion of a multiplexer runs at system interrupt time, the module containing the interrupt entry must be wired. This means not only that its entry in the MST header must specify the wired attribute, but that is must not reference unwired databases (such as error_table_). In addition, any entry that can be called at interrupt time must also be wired and be capable of wiring its stack if necessary. In particular, if the interrupt handler generates send_output interrupts, the write entry must be prepared to run at interrupt time, since tty_interrupt sometimes calls channel_manager$write in response to such interrupts.

In order to avoid excessive system interrupt overhead, extensive data transformations at interrupt time should be avoided if possible. As noted above, the necessity for extensive transformations of input can be avoided by the use of input_available (rather than accept_input) interrupts, thereby postponing data transformation until the read entry of the multiplexer is called.

MULTIPLEXER INITIALIZATION

At answering service startup time, the initializer process initializes each configured FNP (or other root-level multiplexer) that is specified (and not marked inactive) in the CDT. This process includes initializing each active, configured subchannel of such a multiplexer, and so on down to the level of the nonmultiplexed channel. (If a multiplexer is marked inactive in the CDT, it is not initialized automatically, but may later be initialized manually by means of the load_mpx operator command.)

The process of initializing a multiplexer may be regarded as having three stages: database initialization, loading, and subchannel initialization.

Database Initialization

Database initialization is accomplished by a call to the init_multiplexer.entry of the multiplexer module, as described earlier in this section. The initializer calls hphcs_$init_multiplexer, which call is forwarded to priv_chanel_manager$init_multiplexer, which in turn calls the

multiplexer module through cmtv. The init_multiplexer entry is responsible for allocating and initializing any databases needed by the multiplexer module.

Loading

The loading of each type of multiplexer is handled by an answering service (user-ring) module named as_<TYPE>_mpx_, where <TYPE> is the name of the multiplexer type. This module contains two entries, named load and dump (see below). The load entry makes whatever calls are necessary to the supervisor in order to accomplish physical initialization of the multiplexer connection. This may involve calling a special supervisor loading module, or (more likely) calling the hpriv_control entry of the multiplexer module (through hphcs_$hpriv_control) with a control operation used for loading. The loading process may be as simple as issuing a "listen" control operation on the multiplexer channel and waiting for a dialup interrupt, or it may involve a handshake sequence with the channel, depending on the type of communications protocol being used. In either case, once the connection is established, the multiplexer module must inform the initializer by means of a wakeup over an event channel whose name is passed on to ring 0 at load time. If the load fails, the initializer must be similarly informed. The result of the load attempt is reflected in the event message associated with the wakeup.

Subchannel Initialization

Once the initializer has been informed that a multiplexer has been loaded successfully, it proceeds to initialize all subchannels of the multiplexer, by calling hphcs_$init_channel (for nonmultiplexed channels) or hphcs_$init_multiplexer (for multiplexed channels) for each one. The loading process is also reiterated for each multiplexed subchannel.

CRASHING AND REINITIALIZATION

If the multiplexer module detects that the multiplexer channel has "crashed", i.e, hung up or otherwise become unusable, it takes the following steps:

1. Send a crash interrupt to each currently active subchannel;

2. Send a wakeup to the initializer (over the event channel provided at load time) to notify it that the multiplexer is down.

On receiving this wakeup, the initializer attempts to take a dump of the multiplexer by calling the as_<TYPE>_mpx_$dump entry in the multiplexer-specific answering service module mentioned under "Initialization" above. (If no dumping mechanism exists, the entry should simply return.) The initializer then calls the shutdown entry of the multiplexer to clean up and deallocate its databases, and then reinitializes it as described above.

EXAMPLES OF CALL AND INTERRUPT PATHS

Figure 3-1 shows some possible paths of calls to Multics Communication System to send output to various channels. Each path in and out of channel_manager is labeled according to the channels on whose behalf the call is being made. The boxes marked with an asterisk represent hypothetical multiplexers; the example shows both a multiplexed subchannel of an FNP (called "intermediate_mpx") and a site-specific substitute for the DN6600/6670 (called "other_fnp"). The paths for other calls (such as start or control) are similar. Note that under some circumstances a given level of multiplexer might not pass a call on to the next level.

Figure 3-2 shows the paths taken by interrupts from various channels on the same configuration as in Figure 3-1. Once again, the paths into and out of channel_manager are labeled in order to show the paths taken by interrupts from specific channels.

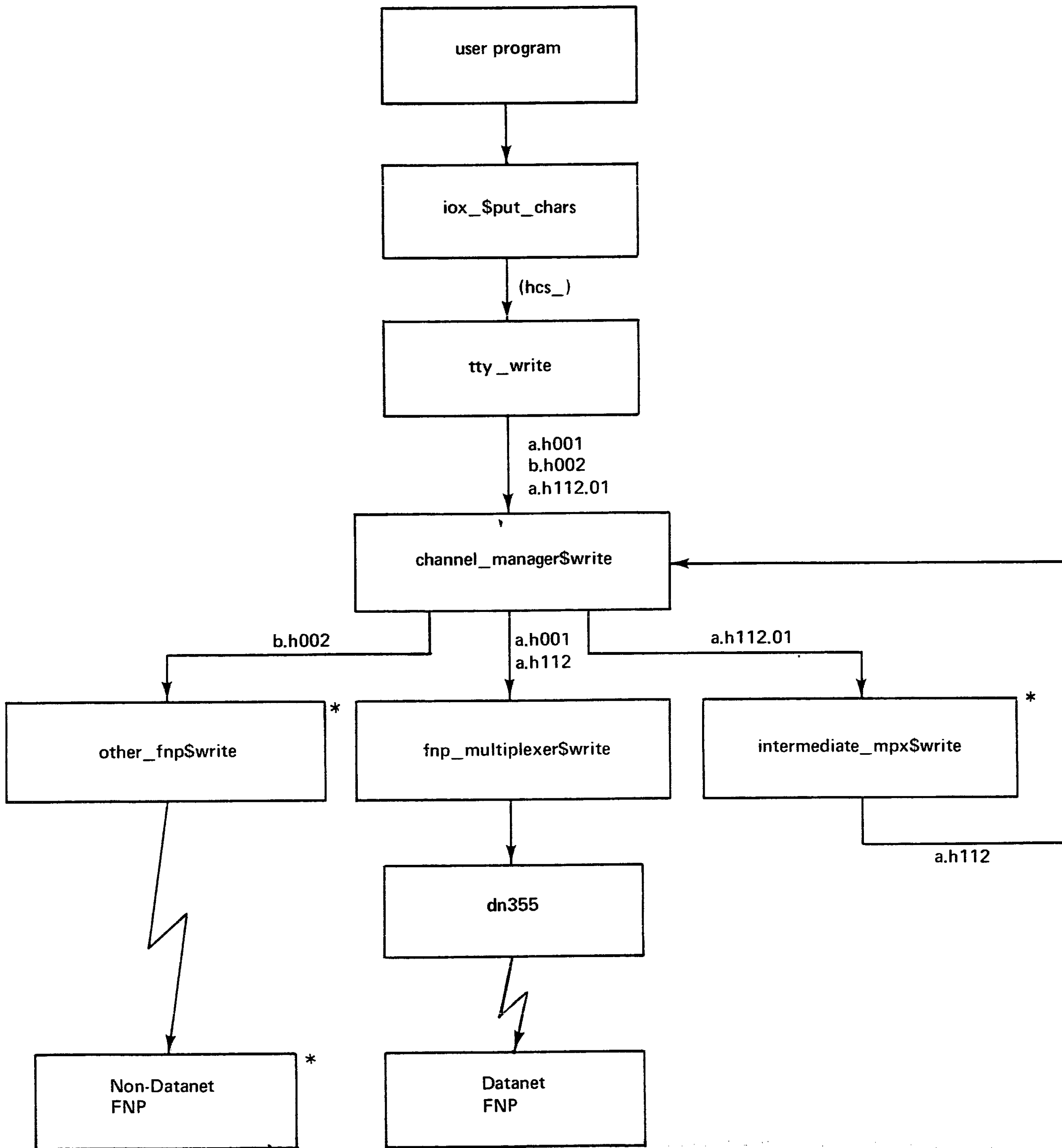


Figure 3-1. Possible Paths of write Call

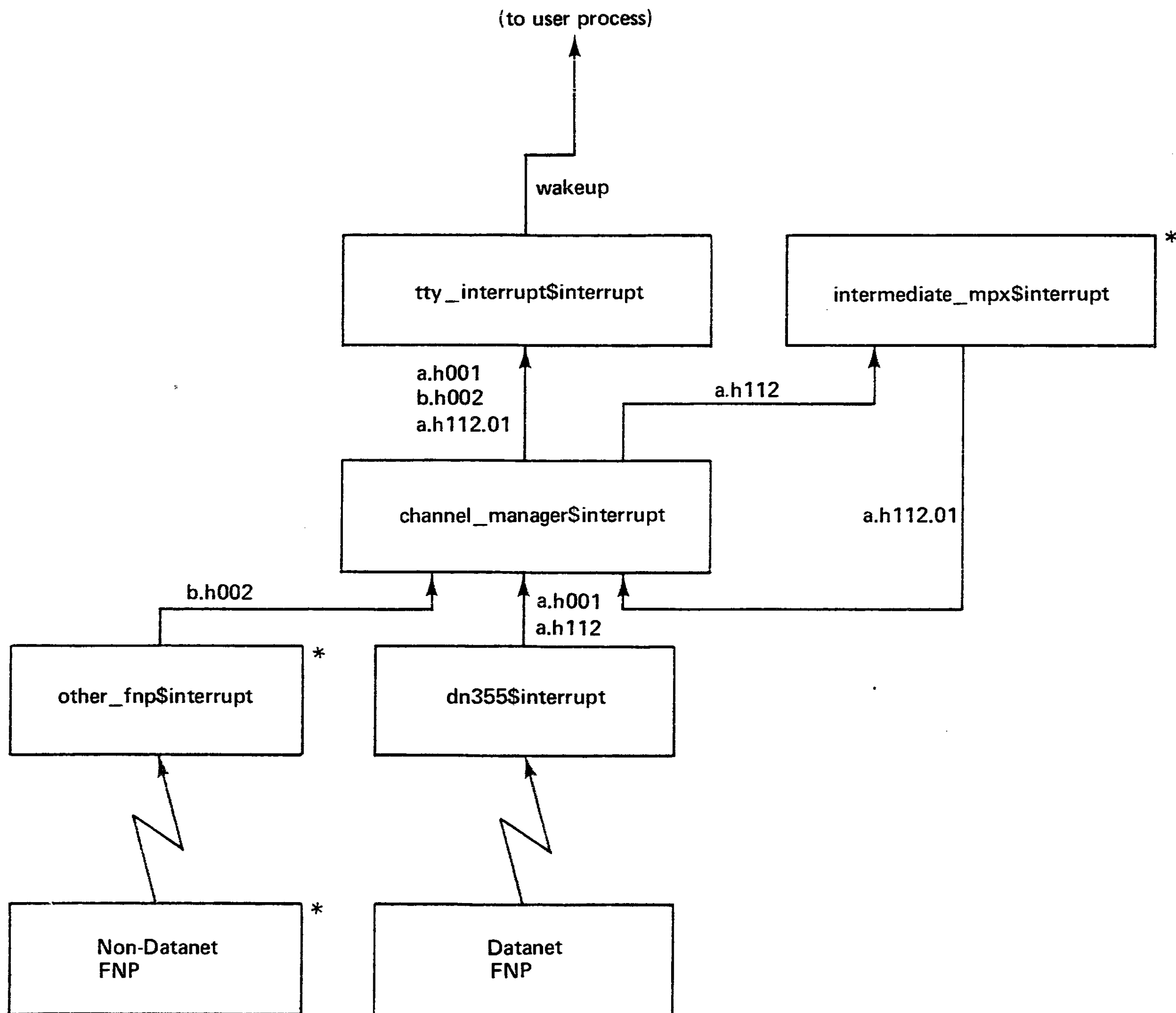


Figure 3-2. Possible Paths of Interrupt

SECTION 4

FNP INTERFACE MODULE

All communication with the FNP over the Direct Interface Adapter (DIA) is managed by the dn355 module. Control information is passed between the two computers by means of the mailbox segment, dn355_mailbox. A pointer to the mailbox area used for each FNP is set in dn355_data at system initialization time by fnp_init. The mailbox area is described in Section 2.

When Multics Communication System was originally designed, one of the design goals was compatibility with another FNP software package called NPS. This goal was subsequently abandoned, but some of the compatibility features remain; this is the source of some peculiarities in the FNP/CS interface, such as the use of some apparently unnecessary mailbox operation codes and the presence of some unused fields in the submailbox.

CALLS AND INTERRUPTS

There are four entry points to dn355: dn355$send_wcd and dn355$send_wcd_global, which are called by fnp_multiplexer to send output or control information to the FNP; dn355$interrupt, which is invoked when an interrupt arrives from the FNP; and dn355$hangup_fnp_lines, which is called when an FNP is shut down.

The normal "called" entry (dn355$send_wcd) is passed a mailbox operation code and optional command data, which is copied into the lowest-numbered available submailbox to be sent to the FNP; it then calls the internal procedure send_mbx to interrupt the FNP at the level corresponding to the submailbox. If the operation being performed is the transmission of output data, some more processing is required before sending the mailbox, as described later in this section. The send_wcd_global entry is used for control information that is not specific to a particular channel.

An interrupt comes from the FNP under either of the following circumstances:

- the FNP has filled in or completed processing of a submailbox, and wants the CS to look at that submailbox;

- the FNP has crashed and sent an "emergency" interrupt.

If the FNP has crashed, the interrupt level (passed to dn355$interrupt in its fourth argument) is 7; in this case, words 6 and 7 of the mailbox header describe the cause of the crash. If the interrupt level is 3 (for a "normal" FNP interrupt), dn355 checks the "terminate interrupt multiplex word" (TIMW) in the mailbox header to see if the FNP has processed a submailbox, as explained in more detail later in this section.

It should be noted that, each time dn355$interrupt runs, there may have been more than one interrupt from the FNP since the last time it ran; for instance, interrupts may have been masked when the FNP interrupt occurred (dn355$send_wcd usually runs with interrupts masked). Accordingly, dn355$interrupt takes care of all pending business (submailboxes processed by the FNP) every time it runs.

MAILBOX TRANSACTIONS

Transactions Initiated by the FNP

FNP-CONTROLLED MAILBOXES

Mailboxes 8 through 11 are reserved for FNP-initiated transactions; it is the responsibility of the FNP to know which of these mailboxes are currently available. Whenever the FNP needs to initiate a transaction with the CS, it selects one of these mailboxes and fills it in with the FNP line number of the relevant channel, an RCD (read control data) I/O command, the operation code describing the particular transaction, and any associated command data. It then writes the submailbox into CS memory, turns on the bit in the TIMW in the mailbox header corresponding to the submailbox, and sends an interrupt to the CS.

PROCESSING THE RCD

When dn355 runs as a result of the abovementioned interrupt it examines any submailboxes corresponding to 1-bits in the TIMW. In the case of the mailbox used in the preceding paragraphs, it sees the RCD I/O command and knows that it must process the operation code supplied by the FNP. For most operation codes, such as "accept new terminal", "disconnected line", or "break

condition", this processing is simple and straightforward; it consists primarily of forwarding the interrupt to channel_manager$interrupt, and in some cases rewriting the mailbox with an appropriate acknowledging operation code. The interrupt is eventually forwarded to tty_interrupt, which sends a wakeup to the relevant process to inform it of the change in the channel's state. The FNP is notified that the CS has processed the mailbox by means of an interrupt; if the interrupt is at the level corresponding to the mailbox number (8 to 11), it means that dn355 has updated the contents of the mailbox, and the FNP must read it back; an interrupt at a level 4 greater than the mailbox (i.e., 12 to 15) means that the mailbox is free for further use.

There are three RCD operation codes that require a more extended transaction between the FNP and the CS: "send_output", "input_in_mailbox", and "accept_direct_input." These operation codes are used as "carriers" for reporting the amount of buffer space currently available in the FNP; this value is copied from the command data by dn355 and kept both for metering purposes and to determine how much output to send at a time.

send_output OPERATION CODE

The "send_output" operation code is sent by the FNP when it is ready to accept output for a particular channel. When dn355 finds this operation code in a submailbox, it checks to see if there is any output for that channel waiting in tty_buf; if there is not, it forwards a send_output interrupt through channel_manager to tty_interrupt, which either calls back with any pending output that it did not send before, or wakes up the user process. If there is output waiting, dn355 prepares to send a WTX (write text) submailbox, as described later under "Transactions Initiated by the CS."

SHORT INPUT -- input_in_mailbox

The "input_in_mailbox" operation code indicates that the FNP has 100 characters or less of input for the channel identified in the submailbox. The input itself is contained in the submailbox. If there is enough free space in tty_buf to copy this input into a buffer, dn355 allocates such a buffer, copies the data into it, and forwards an "accept_input" interrupt. It then sends the FNP an interrupt to free the submailbox. If buffer space cannot be obtained, dn355 rewrites the mailbox with a "reject request" operation code. In response to this code, the FNP retries the "input_in_mailbox" one second later.

LONG INPUT -- accept_direct_input

The "accept_direct_input" operation code indicates that the FNP has more than 100 characters of input for the channel identified in the submailbox. This input will be written by the FNP into the circular buffer in the header of tty_buf. Accordingly, dn355 checks to see if there is sufficient space in the circular buffer for the specified number of characters; if there is not, or if it appears that there will not be enough space in tty_buf to build an input chain, dn355 rewrites the submailbox with the "reject request" operation code as above.

If there is sufficient space, dn355 fills in the submailbox with an RTX (read text) I/O command, an "input accepted" operation code, the absolute address pointed to by the circular buffer free pointer (tty_buf.cq_next), and the character count. If the data will not all fit between cq_next and the end of the circular buffer, the remaining data is put at the beginning of the buffer, and a "wraparound" address and character count are also provided in the submailbox. The free pointer (cq_next) is updated to point past the last character that will be input, the free size indicator (tty_buf.cq_free) is decremented accordingly, and the submailbox is sent to the FNP.

The FNP responds to the RTX submailbox by writing the input into the designated portion(s) of the circular buffer and setting the TIMW bit for the submailbox so that dn355 will look at it again. When dn355 sees the RTX, it allocates a sufficient number of buffers to hold the input, and chains them to the channel's input chain (or starts an input chain if one does not already exist for the channel). Any space left at the end of the last buffer in an existing input chain is used before additional buffers are allocated. The data is then copied from the circular buffer into the input chain, cq_free is incremented by the amount of space thus freed in the circular buffer, an interrupt is sent to the FNP to free the submailbox, and an "accept_input" interrupt is forwarded as above.

HANDLING THE accept_input INTERRUPT

If the input contains a break character (as indicated by a flag in the interrupt data) and the using process has tried unsuccessfully to read data from the channel (as indicated by wtcb.rflag), tty_interrupt sends a wakeup to the process so that it will call tty_read again and pick up the input. (A wakeup is also sent if the process is waiting for the terminal's answerback, as indicated by wtcb.wru.) An exception to this arises when the break character is a formfeed; in this case, dn355 does not forward an "accept_input" interrupt. If output to the channel had been suspended because of a full-page condition, dn355 checks to see if the PCB points to a pending output chain; if it does not, a "send_output" interrupt is forwarded;

otherwise, an entry is added to dn355's delay queue so that output processing will be resumed when dn355 finishes dealing with the input. (See also the description of the delay queue mechanism later in this section.)

Transactions Initiated by the CS

When tty_write or fnp_multiplexer wants to send output or control information respectively to the FNP, a call is made to fnp_multiplexer, which in turn calls dn355$send_wcd or dn355$send_wcd_global. The calling sequence of these entry points includes the operation code to be sent to the FNP and the associated command data, if any. When called at either of these entries, dn355 scans the "used" flags in the mailbox header looking for an unused submailbox. If all the submailboxes are currently in use, it calls the internal procedure make_q_entry to create an entry for the transaction in the delay queue, and returns; otherwise, it fills in the first free submailbox it finds and sends it to the FNP.

When passing control information, dn355 simply copies the operation code and command data into the submailbox, sets the line number in accordance with the device index supplied by the caller, sets the submailbox I/O command to WCD (write control data), and calls iom_manager to interrupt the FNP at the appropriate level. When the FNP has read the submailbox, it sets the corresponding TIMW bit and sends an interrupt; dn355$interrupt then sees the interrupt for a WCD submailbox and, knowing that the submailbox has been processed by the FNP, simply marks it as free (by turning off the corresponding "used" flag). More work has to be done, however, if the operation code is "accept direct output," indicating that there is output data to be sent to the FNP. The processing of output data by dn355 is described below.

OUTPUT DATA

The internal procedure process_send_output is invoked in response to either a "send_output" operation code sent from the FNP or a call to dn355$send_wcd with an "accept_direct_output" operation code. It ensures that there is in fact an output chain for the current channel, and that output to that channel has not been suspended because of a full-page condition; assuming both tests succeed, a block is allocated in tty_buf in which an array of "pseudo-DCWs" describing the output is built. Each pseudo-DCW contains the absolute address and character tally of one buffer in the output chain; the number of pseudo-DCWs constructed, and hence the number of buffers of output that are sent to the FNP in one transaction, depends on the amount of buffer space remaining in the FNP and the actual size of the chain, but in no case may exceed 16, which is the number of pseudo-DCWs that fit in one block. If dn355, while scanning the output chain, encounters a

buffer with its end_of_page flag on, that buffer is the last one to be sent in this transaction, and pcb.end_frame is turned on so that the remainder of the chain (if any) will be processed when, and only when, a formfeed is input from the channel (see "Input Data," above).

Once the pseudo-DCWs have been built, the absolute address of the pseudo-DCW block and the number of pseudo-DCWs is put in a submailbox, along with a WTX (write text) I/O command and an "accept_direct_output" operation code (as a matter of NPS compatibility, the operation code is "accept_last_output" if the last buffer in the output chain is being sent, but the FNP makes no distinction between these two codes). This submailbox is then sent to the FNP. If the last buffer in the output chain is being sent in this transaction, and dn355 was entered through the interrupt entry, a "send_output" interrupt is sent to channel_manager. On receipt of this interrupt, tty_interrupt may send the next block of output if one is available, or send a wakeup to the user process so that tty_write can be called again. The forward pointer in the last buffer transmitted is set to zero so that any remaining buffers in the output chain will not be freed when the output completes (see below).

When the FNP reads the submailbox containing the WTX, it reads the pseudo-DCW block and then uses the pseudo-DCWs to read the actual data into FNP memory. Once it has done this, it sets the TIMW bit corresponding to the submailbox; when dn355 examines the submailbox containing the WTX, it frees the portion of the output chain that was transmitted and the pseudo-DCW block.

GLOBAL OPERATIONS

A few of the control operations sent to the FNP do not refer to any particular channel but rather to the FNP itself: "accept_calls," "dont_accept_calls," "dump_fnp," and "patch_fnp." These are treated like other control operations, except that dn355 is called at the send_wcd_global entry rather than send_wcd. The line number field in the submailbox is accordingly set to zero. For dump_fnp and patch_fnp, used to dump and patch specified portions of FNP memory, respectively, fnp_multiplexer has called pxss$wait so that it can be informed when the I/O is complete; therefore, when the WCD submailbox is freed, if it contains either of these operation codes, dn355$interrupt calls pxss$notify.

LOCKS

The channel lock in the LCTE for an FNP is used to protect not only the LCTE, but also the fnp_info structure, all PCBs, and the mailbox area for that FNP from simultaneous access by more than one processor. Because these data bases are accessed at

interrupt time before channel_manager is invoked, dn355 and fnp_multiplexer must manage the lock themselves, instead of relying on tty_lock as described in Section 6. (A flag in the LCTE informs channel_manager that it is not to call tty_lock for an FNP channel.) Accordingly, dn355$interrupt and most entries in fnp_multiplexer loop on the channel lock shortly after being called. The fnp_multiplexer entries must call pmut$wire_and_mask before doing this so as not to lose the processor with the FNP channel locked; they also check to see if they have been called as a result of an interrupt, in which case the channel has already been locked by dn355.

In addition, the circular buffer must be locked when the relevant pointers and free space indicators are being updated, to prevent two or more processors from attempting to modify it simultaneously on behalf of different FNPs. This situation is not covered by the fact that the circular buffer is never accessed without the FNP channel being locked because there is only one circular buffer for the whole system, rather than one per FNP.

DELAY QUEUING

If a mailbox transaction could not be performed when it was requested because no submailbox was available, an entry is added to a "delay queue" allocated in tty_buf so that the requested processing can be performed later. There is a separate delay queue for each FNP. An entry in this queue contains the offset of the PCB for the relevant channel, the mailbox operation code, and any associated command data. If the entry is for a global FNP operation, the PCB offset is 0.

The dn355$interrupt entry checks to see if there are any entries in the current FNP's queue before and after examining the TIMW; if there are, it calls an internal procedure named process_q, which performs the operation requested by each queue entry as long as there are submailboxes available.

SECTION 5

INTERFACES TO THE USER RING

This section describes the hardcore portions of Multics Communication System that are called from outer rings in user processes. There are three modules that can be called thus:

1. tty_write
   processes user-supplied output, copies it into tty_buf, and passes it on through dn355 to the FNP. It is called through the hcs_$tty_write and hphcs_$tty_write_force gate entries.

2. tty_read
   converts terminal input from tty_buf and copies it into a user-supplied buffer. It is called through the hcs_$tty_read and hcs_$tty_get_line gate entries.

3. tty_index
   performs a variety of user-requested utility functions, principally the processing of control operations. It is called at a variety of entry points through several gate entries, discussed later in this section.

The precise calling sequences of these entry points can be ascertained by looking at the source code. In general, neither user nor system programs call these gate entries directly, but go through the I/O system (iox_) which calls entries in the user-ring typewriter I/O module (tty_). For details of this mechanism, see the <u>User Ring Input/Output</u> PLM, Order No. AN57.

Both the tty_write and tty_read modules make use of an ALM subroutine tty_util_, which uses EIS instructions to do translations and searches of the input and output character strings. In addition, tty_read calls tty_canon to put input strings in canonical form (for a complete description of Multics canonical form, see the <u>MPM Communications Input/Output</u>, Order No. CC92). The tty_index module calls tty_modes to perform a "modes" control operation. All three modules use tty_tables to select translation and special-characters tables.

The TCB contains relative pointers to the various tables used in the conversion of input and output. These pointers are set to zero when the channel is initialized, and subsequently set to default values by the "set_terminal_data" order. They may subsequently be changed as a result of control operations such as set_delays, set_output_translation, etc.; they are always restored to the default values when the "using" process is changed. The pointers used for output conversion by tty_write are copied into automatic storage.

OUTPUT CONVERSION - tty_write

The tty_write module takes user-supplied ASCII characters and converts them to a form suitable for printing on the user's terminal. Before it begins, it ensures that the device index passed to it is valid, that it refers to a terminal that is currently dialed up, and that the user process is actually using that terminal (i.e., wtcb.uproc contains the current process' id). It locks the LCTE to ensure that the terminal's state does not change out from under it (unless it was called at the tty_write$locked entry, which is called with the LCTE already locked by tty_index to write printer_on and printer_off sequences).

The caller provides the count of characters that are to be processed ("nelem") and tty_write returns the number that were actually processed ("nelemt"). If nelemt is less than nelem, it is assumed that the caller will go blocked on the event channel named by wtcb.event; to signify that this is the case, tty_write turns wtcb.wflag on. On finding the flag on, tty_interrupt sends a wakeup over the event channel when a send_output interrupt comes from the FNP, as described in Section 4.

The functions of tty_write can be logically divided into four phases:

1. Preliminary conversion, as in the translation of lowercase letters to uppercase for a Teletype Model 33 or terminals in "capo" mode;

2. Formatting, e.g., substitution of escape sequences, insertion of newline characters in long lines, canonicalization and optimization of white space, etc.;

3. Translation, as from ASCII to EBCDIC;

4. Buffer allocation and copying of characters into buffers in tty_buf, from which they are read by the FNP.

Each phase is executed over the entire input string (or as much of it as is transmitted at once) before passing on to the next phase. In most cases, of course, either phase 1 or phase 3 or both can be omitted; in "rawo" mode, tty_write can and does proceed directly to phase 4.

Each phase is provided with an "input pointer" to the location where the previous phase left the data in its latest form. This pointer points either to the user's original input or to either of two buffers in the automatic storage of tty_write, as described later.

The rest of this section contains a more detailed description of the four phases of conversion mentioned above and a discussion of space allocation and character counting.

Preliminary Conversion

Certain terminals require uppercase-only output; similarly, a user can specify (by entering "capo" mode) that all lowercase letters are to be converted to uppercase for output. These cases are treated identically by tty_write: an mvt (move with translation) instruction is used to copy the user's data into an automatic buffer, using a translation table that substitutes uppercase ASCII for lowercase. If the user is in "edited" mode, this is all that needs to be done for this phase; if not, however, each letter that was originally uppercase must be preceded by an escape character ("\"). Therefore, in "^edited" mode, the translation table also replaces each uppercase letter with the same character with its high-order bit (the "400(8)" bit) turned on. After the mvt is completed, an scm (scan with mask) instruction is executed to find the first character with the "400" bit on; if one is found, all characters to the left of it are copied to a second internal buffer, an escape is inserted after the copied characters, and the high-order bit of the found character is turned off. The scm is repeated on the remainder of the characters in the first buffer until all characters have been copied to the second buffer with escapes inserted as needed. If no characters with the high-order bit on are found in the entire string, no copying is done.

Formatting

The search for, and correct handling of, "interesting" characters is the most crucial function of tty_write, and the one to which most of the time spent in tty_write is devoted. The identification of "interesting" characters is facilitated by the use of the tct (test character and translate) instruction under control of the output conversion table, which contains zero entries for all "uninteresting" characters and various indicators identifying the different kinds of "interesting" ones: carriage movement characters, ribbon shifts, and characters requiring the substitution of escape sequences.

The formatting phase of tty_write calls tty_util_$find_char to find the first "interesting" character in the string; tty_util_$find_char returns a tally of "uninteresting" characters skipped over, the indicator value for the character it stopped at, and an updated pointer to the character at which to start the next scan. The tty_write module copies the uninteresting characters into an internal buffer (whichever one does not contain the source string) and examines the indicator. If it designates an escape sequence, the sequence is inserted in the buffer. For a newline, vertical tab, or formfeed character, the special_chars table is indexed to find the appropriate representation of the character, and the delay table is searched to find the correct number of delays to be inserted depending on column position, terminal type, and baud rate. For "white space" (horizontal tab, backspace, carriage return, or two or more blanks) tty_write simply calculates and remembers what column position to end up in; this information is either used to insert appropriate carriage motion characters before the next graphic to be inserted, or discarded if the next character involves vertical carriage motion. This process is repeated until all the source characters are used up. If it happens that the first call to tty_util_$find_char returns an indicator of zero and has used up the entire source string, no characters are moved by this phase.

Another responsibility of the formatting phase is the counting of output lines and watching for full pages. When the line count reaches maximum, the formatting phase inserts a warning string (such as "EOP") and a sentinel character at the end of the page, and the copying phase (see below) later removes each sentinel and turns on a flag in the buffer that ends the page. When dn355 sees this flag, it ceases transmission, and sets pcb.end_frame. When it receives input for a channel with pcb.end_frame on, it scans this input for a formfeed; if it finds one, it turns off the flag and starts up output for the channel again.

Translation

The translation phase is very similar to the preliminary conversion phase described earlier. An mvt instruction is used to copy the entire string from wherever it was left by the preceding phase to an automatic buffer, translating it from ASCII to the appropriate output code in the process. This does not complete the process for a terminal which requires case-shift characters (which currently includes most terminals for which translation is done); the insertion of case-shift characters is done in a similar manner to the insertion of escapes before capital letters as described under "Preliminary Conversion." The translation table causes the high-order bit of each uppercase character to be turned on (in this context, the term uppercase refers not only to capital letters but to all characters for which the shift key must be depressed while typing) and the "200(8)" bit of each lowercase character to be turned on; characters that may be in either case (such as space) contain no extra bits. After translation, an scm is done to find the first character in the opposite case to the one in which the terminal was at the start of the output; all characters to the left of it are copied, an appropriate shift character is inserted after the copied characters, and another scm is used to find the next change of case. If all the output characters are in the same case, no copying is done. Note that it is not necessary to turn off the high-order bits of the uppercase characters, since these bits are ignored by the remainder of Multics Communication System and ultimately thrown away by the FNP.

Buffer Allocation and Copying

The final phase of tty_write consists of allocating buffers in tty_buf and copying the final output into these buffers. The maximum buffer size for a channel is derived from its baud rate -- the faster the channel, the larger the buffers it gets. The size of buffer actually allocated for a given output message is the smallest multiple of 16 words in which the entire message will fit; if the entire message does not fit in the channel's maximum-size buffer, additional buffers are allocated and chained on to the first one. If an end-of-page sentinel is encountered, a flag is turned on in the current buffer, and the buffer is not filled past the sentinel. If output already processed for the particular channel has not yet been sent, a chain of buffers for that channel already exists, starting at the offset in wtcb.write_first; if the last buffer in this chain is not full, and does not have its end-of-page flag on, it is filled before further buffers are allocated. If the last buffer is less than the maximum size, it is replaced (if possible) by a larger buffer in which is placed the contents of the original buffer and as much of the new output will fit. The newly-allocated buffers are threaded onto the old chain. If wtcb.write_first is zero, tty_write starts a new chain. Finally, if wtcb.send_output is on, indicating that the parent multiplexer is prepared to handle output for the channel, tty_write calls channel_manager$write to

forward the output chain (or as much of it as possible) to the multiplexer.

Space Allocation and Character Counting

Because the input string undergoes wholesale modification at several points, it is necessary to decide how many of the user's characters to process before actually doing anything. Certain constraints are applied to keep any one channel from monopolizing tty_buf: no more than a certain fraction of available buffers in tty_buf are to be assigned to a single channel at any time; and no output chain of more than a certain number of buffers is built. The particular numbers involved are, for the sake of convenience and simplicity, preset system-wide constants. The current values are 1/4 and 16 respectively; i.e., no channel is ever assigned more than 1/4 as many buffers as are free at the time of assignment, or more than 16 buffers is a single output chain.

This restriction does not apply when tty_write is entered at tty_write_force; in this case, the channel can have as many buffers as it needs as long as at least 32 words are left over.

The number of characters to process may then be expressed as:

```
nchars = min(chars_supplied, maxbuf*chars_per_buffer)
```

If the terminal is in "rawo" mode, this is the number of characters that are actually shipped, and nothing further need be done. In general, however, the number of characters actually output is somewhat larger than the number supplied; meters done at various times show an average growth ratio of about 6:5. Accordingly, for nonraw output, tty_write multiplies nchars as calculated above by 0.8 to allow for growth (this actually allows for a growth ratio of 5:4, giving us some leeway). As a result, the size of the output string can grow by as much as 25% without requiring more buffers than one channel is "supposed" to have; however, the restriction to 1/4 of the available buffers is a very conservative one, so if it occasionally proves necessary to allocate an "extra" buffer, the overall effect on available buffer space should not be noticeable.

An additional consideration arises from the use of internal buffers in tty_write. Because of the possibility of more than one intermediate copy, two such buffers are needed, and rather than create two segments so as to allow each buffer to grow essentially without limit, it was decided to set aside fixed-size buffers in the stack frame of tty_write. The size chosen for each of these buffers is the maximum allowable output chain size.

Clearly growth ratios greater than 5:4 can and do occur; there are pathological cases such as an object or other non-ASCII segment being printed on a 2741 terminal, which involves a growth ratio of more than 6:1 (<upper_shift> ¢ <lower_shift> nnn for each input character, plus added newlines and ¢c markers). Thus, despite precautions, tty_write must be prepared for the possibility that in the course of translation or formatting it will run out of space in the internal buffer. When this happens, the number of input characters to be handled is cut in half, and character processing is started over from phase 1.

If space in tty_buf is unusually tight, then an abnormal character string that is not large enough to overflow the internal buffer space might nonetheless require the allocation of more buffers than are available. If tty_write finds that it is about to allocate the last buffer, it takes the same action as if it were about to overflow one of its internal buffers, i.e., divide the number of input characters in half and start over. If this happens often, it is probably an indication that tty_buf is too small, and its size as defined on the PARM TTYB of the configuration deck card should be increased.

If no buffers at all can be obtained, either because there are none available or because the channel already has as many as it is entitled to, no output is processed and zero is returned in nelemt. If, in addition, wtcb.send_output is on, tty_space_man$needs_space is called to ensure that the process receives a wakeup when more buffers become available (see the discussion of tty_space_man in Section 6). If wtcb.send_output is off, it is sufficient to turn on wtcb.wflag; this ensures that the process receives a wakeup when the multiplexer next requests output.

INPUT CONVERSION - tty_read

The tty_read module takes typed input characters from the specified channel's input chain (pointed to by wtcb.fblock) and copies them, after suitable conversion, to a buffer supplied by the caller. There are two entries to tty_read that are normally called: tty_read itself and tty_read$tty_get_line. The former is called as a result of a get_chars operation, the latter as a result of a get_line operation. The difference is that the get_line entry only returns characters up to and including the first available newline character. Like tty_write, tty_read first validates the device index and ensures that it corresponds to an active channel, as well as locking the LCTE. Also like tty_write, it takes an input argument ("nelem") specifying the size of the caller's buffer and returns an output argument ("nelemt") specifying the actual number of characters copied into the caller's buffer. nelemt is the smallest of the following:

- nelem;

- the total number of characters (after conversion) in the read chain;

- if tty_get_line was called, the number of characters (after conversion) up to and including the first newline character in the read chain.

If nelemt is zero, it is assumed that the caller will go blocked on the event channel whose name is in wtcb.event; tty_read accordingly turns wtcb.rflag on when returning zero in nelemt. When input containing a break character is copied into tty_buf by dn355 and forwarded to tty_interrupt (as described in Section 4), if wtcb.rflag is on, tty_interrupt sends a wakeup over the event channel so that the user process can call tty_read again.

The break character recognized by tty_read is determined according to the line type associated with the channel. In general, the break character is a newline character.

Certain transformations may be performed on the characters typed by the user, such as reduction to canonical form, removal of "erased" and "killed" characters, and the interpretation of escape sequences. The application of these transformations depends on both the modes associated with the channel and the contents of the relevant tables in tty_tables.

The functions of tty_read may be divided into the following phases:

1. <u>Copying</u> raw input data from tty_buf, and freeing the ring 0 buffers;

2. <u>Translation</u> to ASCII

3. <u>Canonicalization</u> of the contents of column positions

4. <u>Erase and kill processing</u>

5. <u>Escape sequence processing</u>

Clearly, these five phases are not always necessary. Phases 3, 4, and 5 depend on "can," "erkl," and "esc" modes, respectively; in "rawi" mode, only phase 1 is required.

For convenience and to ensure consistency, conversion (the generic term used here for the relevant subset of phases 2

through 5) is done on all characters up to and including the first break character in the input chain, whether or not the break character is found within the limit specified by the caller. This avoids the possibility of terminating conversion in the middle of an escape sequence or a line that is subsequently killed, and also allows for the possible shrinkage of the input string (through the deletion of extraneous white space and the condensation of escape sequences, for example). "Extra" characters thus converted (i.e., those that cannot be returned because the caller has not provided sufficient space) are saved in reallocated buffers in tty_buf; these buffers are marked by turning on buffer.converted and chained to the head of the channel's input chain so that they can be picked up by the next call to tty_read. In two exceptional cases, conversion cannot proceed to the first break character: the first is, obviously, when no break character is present; the other is when the size of the internal automatic buffers of tty_read is exceeded.

The remainder of this discussion consists of a few remarks on the management of the internal buffer of tty_read and a more detailed description of the five conversion phases mentioned above.

Space Management

During conversion, intermediate forms of the input string result from each conversion phase; for the storage of these intermediate strings, two buffers are maintained in the automatic storage of tty_read. Clearly this sets an upper limit on the allowable length of the input string. The normal limiting factor, of course, is the presence of a break character, and input lines longer than 100 characters are rare; a further limitation is imposed by the FNP software, which takes a channel out of receive mode if more than 600 characters are input without a break character, causing "exhaust" status in the FNP (see Section 12). The input string can grow during canonicalization through the replacement of carriage returns by multiple backspaces, but this occurrence too is rare. All in all, a buffer size of 720 is very unlikely to be exceeded.

Consequently, no more than 720 characters are copied into the internal buffer from tty_buf. If the canonicalization phase attempts to increase the length of the string past 720, tty_read reduces the limit by one-third and starts again. Because of the possibility that this restart may be necessary, buffers in the read chain from which characters have been copied cannot be freed until after the canonicalization phase is completed.

Since conversion is, if possible, carried out on all characters up to and including the first break character, the final converted string may be larger than the buffer provided by

the caller. If this is the case, enough characters to fill the caller's buffer are returned; the remainder of the converted characters, as indicated above, are saved in buffers in tty_buf in each of which buffer.converted is set. In addition, if one of these buffers contains a break character (the last one generally does), buffer.break is turned on in that buffer. These buffers are added to the head of the input chain as described above.

Copying

IN 'rawi' MODE

The copying phase in "rawi" mode is very simple. Characters are copied from tty_buf, starting at the head of the input chain, directly into the caller's buffer, until either the caller's buffer is filled or the input chain is exhausted. Any buffer from which all the characters are thus copied is freed.

NOT IN 'rawi' MODE

If there are any "converted" buffers at the head of the input chain, characters are copied from these buffers directly into the caller's buffer until either the caller's buffer is full, a break character has been copied, or the chain of converted buffers is exhausted. (In general, the last converted buffer contains a break character, and nonlast converted buffers do not.) Any converted buffer from which all the characters are copied is freed.

If there are no converted buffers, or the converted buffer chain is exhausted without encountering a break character or filling the caller's buffer, characters are copied from the unconverted input chain (if present) into the first automatic buffer of tty_read until either a break character is encountered, the input chain is exhausted, or the internal buffer is filled. Buffers are not freed at this time, for the reason given above under "Space Management."

Because the FNP does not normally send input to the CS until a break character is typed, the input chain almost always ends with a break character. (Consequently, the converted chain usually does, too.) It might not if there was a quit on a channel not in "hndlquit" mode (in "hndlquit" mode the input chain is discarded on a quit), if the channel exceeded the 600-character limit enforced by the FNP software, or if the input is the answerback of the terminal.

If any characters were copied from unconverted buffers, conversion of the contents of the automatic buffer of tty_read begins.

Translation

If a translation table exists for the channel, it is used in a call to tty_util_$mvt to copy the characters from one internal buffer to the other, simultaneously translating it to ASCII. Translation is required for IBM-type terminals using either EBCD or Correspondence character codes; it is also used to translate capital letters to lowercase for uppercase-only terminals such as a Teletype Model 33. (Escaped letters are changed back to uppercase by the escape-processing phase.)

The translation phase does not have to deal with case-shift characters, since the FNP is responsible for recognizing case shifts and for turning on the 100(8) bit in all uppercase characters (characters on shifting terminals are only six bits). All that is necessary in the CS is a translation table that includes characters with the "100" bit on and translates case-shift characters to ASCII NUL characters.

If the channel is not in "ctl_char" mode, a further translation is done using a general table that translates "invisible" characters to NUL (all zero) characters. NUL characters are subsequently discarded by the canonicalization phase. An "invisible" character is any ASCII control character that does not move the carriage or paper, i.e., one that cannot be seen when it is typed. This translation is omitted for terminals such as the IBM Model 2741 and the IBM Model 1050.

Canonicalization

Column-position canonicalization takes care of itself unless the input string contains leftward carriage motion, i.e., backspace and/or carriage return characters. In addition, backspaces and carriage returns at the left margin or immediately preceding a newline are discarded. In other cases, canonicalization must be performed in accordance with the rules given in the MPM Communications I/O.

The canonicalization phase therefore begins by searching the internal buffer (using the PL/I "search" builtin) for a left-motion character (carriage return or backspace). If the first character is a left-motion character, the buffer pointer is advanced by one character, the string length is decremented by one, and the new string is searched as before. If a left-motion character is found, a verify builtin is used to discover if the rest of the line consists of white space (backspaces, carriage

returns, spaces, horizontal tabs, or NULs) followed by a newline. If this is the case, the string length is reduced to the result of the search, and the newline is copied to the new end of the string. If a left-motion character is discovered in any other position, tty_canon is called to perform column canonicalization.

The tty_canon subroutine applies the following algorithm: store each printing graphic from the input string in an array along with its correct column position; sort the array by column position, and by character within each column position; restore the characters to the input string location in the resulting order, inserting backspaces and spaces as appropriate. Tabs must be treated as a slightly special case of printing graphic, so that tabs that are in no way overstruck are preserved but others are replaced by spaces.

The calling sequence of tty_canon has been set up so that the module could theoretically be called with an arbitrary string in other environments than that of ring 0 Multics Communication System. The resulting calling sequence is still not ideal, as it contains arguments that are both input and output; this approach is retained for reasons of efficiency.

The structure used for the elements of the sorting array makes the sort very easy, thus:

```
dcl 1 column_array (max_size) aligned,
      2 column fixed bin (17) unaligned,
      2 erase bit (1) unaligned,
      2 kill bit (1) unaligned,
      2 vertical bit (1) unaligned,
      2 pad bit (5) unaligned,
      2 not_tab bit (1) unaligned,
      2 char char (1) unaligned;
```

The "erase" bit indicates an erase character; the "kill" bit indicates a kill character; the "vertical" bit indicates a nonnewline character requiring vertical carriage motion (i.e., vertical tab or formfeed); the "not_tab" bit is on for any character except a horizontal tab. It can be seen that by treating each element of the array as a single value for the purpose of sorting, the characters automatically come out in column order and in character order in each column, except that: 1) an erase character is always the last character in its column position; 2) a kill character is last in its column position unless overstruck with an erase character; 3) a horizontal tab is always the first character in its column position; and 4) a vertical-motion character follows all characters other than an erase or kill character. Since during the initial scan, a vertical-motion character causes both the "current" column and the "starting" column to be set to the next highest multiple of

1000 (the "starting" column is the column assigned to the left margin, initially 0), a vertical-motion character cannot share a column position unless 1000 or more column positions are actually typed. A newline is assigned a column position of 2**17 - 1 so that it always sorts to the end of the line.

Kill processing is not done by tty_canon; kill characters are sorted to the end of the column position to make things easier for the kill-processing phase of tty_read. Erase characters are only interesting to tty_canon if they are overstruck; since an overstruck erase character sorts to the end of its column position, the rescan step, when it finds an erase character that is not first in its column position, deletes it and all preceding characters with the same column position.

Since a tab sorts to the beginning of its starting column position, it is sufficient to check whether the graphic following the tab has a column position less than the next tab stop; if it does, the tab is dropped, and spaces are inserted as they are whenever there is gap between two graphics. Otherwise the tab is inserted in the final string.

NUL characters are not stored in the column_array; thus tty_canon completes the elimination of "invisible" characters.

The maximum length of the input string is passed as an argument to tty_canon; if the final string exceeds this length, only max_length characters are returned, and a status code of error_table_$long_record is returned.

Upon return from tty_canon, if the status code is zero, tty_read frees the ring 0 buffers from which characters were copied, as explained above; otherwise it reduces its internal buffer size limit by one-third and starts again from the copying phase.

If the canonicalization phase completes without calling tty_canon, the string may still contain NUL characters; therefore if tty_canon has not been called, tty_read indexes the string for NUL characters, and copies the characters preceding and following each NUL into the other internal buffer, decrementing the string length by one for each NUL it finds.

Erase and Kill Processing

Erase and kill processing is really done in two passes, kill and then erase. The string resulting from the canonicalization phase is indexed from the right for a kill character; if one is

found, and the immediately preceding character is not a nonoverstruck escape character, the pointer to the beginning of the string is incremented to point to the character following the kill character, and the length of the string is decremented accordingly. If the kill character is preceded by an escape character that is not preceded by a backspace, the pointer and the length are not changed, and the remainder of the string (if any) is scanned for further kill characters.

The string resulting from the kill pass is now indexed for an erase character. If one is found anywhere but at the beginning of the string, the characters before and after the erased character(s) must be copied to the other internal buffer. The basic mechanism is to copy the characters to the left of the erased characters, decrement the count of total input characters by the number of erased characters plus one for the erase itself, and resume the scan starting with the character after the erase character. (If the erase character is preceded by an escape character not preceded by a backspace, the escape and erase characters are copied along with the preceding characters.) When the end of the string is reached, provided any copying has been done, all characters to the right of the last erase character are copied.

The number of characters to be erased (i.e., not copied) is determined as follows: if the character preceding the erase is "white space" (space or horizontal tab) the source string is searched backward for a nonwhite character, and all characters to the right of it are erased; if the character preceding the erase is a printing graphic, then the source string is searched backward until two nonbackspace characters are found in succession, whereupon all characters from the one to the left of the leftmost backspace on are erased. Note that the character immediately preceding the erase character cannot be a backspace, since all overstruck erase characters are processed by tty_canon.

If the second or subsequent scan turns up an erase character as the first character in the string (as would happen if two erase characters were typed in succession), the determination of the number of erased characters is made in the same fashion as that described above, except that the characters at the end of the target string are examined; the erasing is carried out by decrementing the target pointer so that the erased characters are overwritten, and decrementing the overall length accordingly.

Escape Sequence Processing

This phase, which is implemented in a similar manner to the formatting phase of tty_write as described above, actually deals not only with escape sequences, but with the elimination of white space before break characters and of characters designated as

being "thrown away" for the current terminal type. It uses test character and translate (tct) instructions under control of the input conversion table associated with the channel.

This phase uses tty_util_$tct, which scans for "interesting" characters and returns a tally of characters skipped over, the indicator value for the character stopped at, and an updated pointer to the character stopped at. If the tally is nonzero, tty_read copies the skipped characters into whichever internal buffer does not contain the source string; then it examines the indicator. For a break character, it scans the copied characters (if any) from the right for the last printing graphic; the break character is copied immediately to the right of it. If any intervening white space was found, the length of the final string is decremented by the number of white-space characters. Finally, a flag is set to indicate that a break was found.

If the scan finds a formfeed, and the terminal has a nonzero page length, the formfeed is thrown away, on the assumption that the user typed it for the purpose of starting a new page. Otherwise it is stored as a normal character. The tty_interrupt module is responsible for adjusting the current line count on the page when a formfeed or newline is input.

If the indicator shows an escape character, tty_read must find out if it is in fact the start of an escape sequence. If the channel is not in "esc" mode, or if the character immediately preceding or either of the two characters immediately following the escape character is a backspace, the escape is copied as a normal character and the scan continues. (The backspace test is to ensure that neither the escape nor the column position to its immediate right is overstruck.) If the following character is an escape, erase, or kill character, it is copied to the target string; if it is an octal digit, the character whose value is represented by the one to three nonoverstruck octal digits following the escape character is inserted in the target string; if the escape is followed by zero or more white-space characters followed by a newline, all characters from the escape through the newline are skipped (the newline is not treated as a break in this case); otherwise the character following the escape is looked up in the input_escapes string in the appropriate special_chars structure (described in MPM Communications Input/Output, Order No. CC92). If it is found, the corresponding character from the input_results string is inserted in the target string. If the character is not found, then there is no escape sequence, and the escape character is copied as above. If an escape sequence is identified, the pointer used for the next call to tty_util_$tct is updated to point past the end of the escape sequence.

If the indicator shows that the character is to be thrown away, it is not counted in the length of the final string, and the scan continues starting with the following character. Note that "invisible" characters (see above) have already been thrown away by the time this phase is reached. If the first call to tty_util_$tct returns an indicator of zero and uses up the entire source string, no characters at all are copied by this phase.

If the total number of characters in the now fully-converted string plus the number of previously-converted characters already copied into the caller's buffer is less than or equal to the number of characters requested by the caller, and the converted string ends in a break character, all the converted characters are copied into the caller's buffer, and tty_read returns. If the total number of converted characters exceeds the number requested by the caller, the caller's maximum is copied into the caller's buffer, and the remainder are placed in "converted" buffers in tty_buf as described above, to be picked up by a future call. If the total number of converted characters is less than the number requested by the caller, and the converted string does not end in a break character (either because a break character was escaped, or because the internal buffer size limit was reached), all available characters are copied to the caller's buffer and, if an input chain is still present, the next block of characters (up to the next break) is copied from the input chain and converted as above; any excess characters resulting from the latter conversion are saved in "converted" buffers as above.

Echo Negotiation

A special method of input processing is available to users of breakall mode, intended for use with the experimental emacs editor. This feature is called "echo negotiation"; when it is in effect, although an interrupt is generated by every input character, tty_interrupt only sends wakeups when one of a list of designated characters is encountered. Characters that do not cause wakeups are "echoed" by tty_interrupt, i.e., they are returned to the FNP as output.

The feature is implemented by a combination of a control operation to establish the characters that generate a wakeup, and an entry to tty_read to start echoing. The "set_echo_break_table" order establishes a table of characters in ring 0; once the table has been established, the entry tty_read$echo_negotiate_get_chars may be called to initiate echo negotiation. This entry does everything that the main tty_read entry does; in addition, if it is called at a time when no input for the channel is present in ring 0, it sets a flag in the WTCB. When input arrives and this flag is on, tty_interrupt looks up each character in the table, and either echoes it or wakes up the process. Echo negotiation is turned off by any of the following events: a call to either of the two normal tty_read entries; a

call to the echo_negotiate_get_chars entry that results in characters being returned (in which case a count of the number of characters that have been echoed is also returned); the arrival of a character in the echo break table; or the accumulation of a specified number of characters (which is set by the call to the echo_negotiate_get_chars entry).

UTILITY FUNCTIONS - tty_index

The tty_index module has several entry points that can be called through gates to perform various operations on a nonmultiplexed communications channel on behalf of an outer ring. These functions include initializing, attaching and detaching of channels, passing "ownership" of channels between processes, the implementation of control operations, etc. One other entry, tty_index$initialize_tcb, is called by tty_read and tty_write if they are called before the TCB is initialized. The remainder of this section describes the actions of the different entries.

Initializing a Channel

The init_channel entry is called by priv_channel_manager$init_channel (see Section 3) to initialize a nonmultiplexed channel. It simply allocates a WTCB in tty_buf and a TCB in tty_area and returns a pointer to the WTCB for priv_channel_manager to store in the channel's LCTE.

Terminating a Channel

The terminate_channel entry is similarly called by priv_channel_manager$terminate_channel when a channel's parent multiplexer crashes or hangs up. Its function is to free the space occupied by the TCB and the WTCB.

Assigning a Channel to a Process

The two entries, tty_index and tty_attach, given a channel name, return the device index and the current state of the channel to an outer-ring caller. They are called through the gate entries hcs_$tty_index and hcs_$tty_attach. The device index can then be used in subsequent calls to tty_read, tty_write, and other entries in tty_index. If appropriate, they also make the current process either the "owning" process of the channel (wtcb.hproc) or the "using" process (wtcb.uproc). The two entry points behave in the same way except that tty_attach sets an event channel name in the WTCB, as described later. In effect, a call to tty_attach is equivalent to a call to tty_index followed by a call to tty_event (see below); the following description of the actions of tty_index applies to tty_attach as well.

The first thing tty_index does is to look up the channel name in the LCNT, in order to obtain the device index. It uses this to get a pointer to the LCTE of the channel, which it locks. At this point, if no process has previously been assigned ownership of the channel and the calling process is the initializer, the calling process is made the owning process. If wtcb.hproc is not zero, the calling process' id must already be in either wtcb.hproc or wtcb.uproc; if it is not, an error code is returned and nothing is done. If the caller is entitled to alter the state of the channel (as determined by the above test), the caller's process id is placed in wtcb.uproc (it may have been there already, or the answering service may have taken control of the channel in preparation for creating a new process). If wtcb.uproc is changed by this action, the pointers (in the TCB) to the conversion tables are set to default values. The state of the channel (listening, dialed, or neither) is set according to the flags in the WTCB; then tty_index returns (or, if the tty_attach entry was called, it joins the code for tty_event, described below).

All entries that deal with a specific channel, other than init_channel, terminate_channel, tty_index, and tty_attach, call an internal procedure, setup, that ensures that the device index supplied by the caller is valid, locks the LCTE, gets pointers to the WTCB and the TCB, and checks to see whether the caller is entitled to access to the channel (either the channel is unowned or the caller is either the owner or the user of the channel). It also sets the state of the channel, which is returned to the caller.

Assigning an Event Channel

The tty_event entry is called through the hcs_$tty_event gate entry to record the name of the event channel over which Multics Communication System-originated wakeups are to be sent. After the usual validation, the event channel name supplied by the caller is placed in wtcb.event; this event channel is then used for all output completions, break character inputs, and similar events. If the caller is the owning process, the event channel name is also placed in wtcb.hevent, and is used to signal dialups and hangups as well.

Separating a Channel from a Process

Two entries are used to break the connection between a process and a communications channel: tty_detach and new_proc. The tty_detach entry, called through hcs_$tty_detach, has two uses, depending on the value of its second argument (dflag): if dflag is zero, the caller is presumably the owner, in which case, the channel is taken away from the current user (i.e., wtcb.uproc

is set to zero)--if the user calls this entry, nothing is done; if dflag is nonzero, the caller must be the owner, who is making the call in order to give up ownership of the channel. In this latter case, tty_detach calls channel_manager$control to hang the line up and sets wtcb.hproc to zero. As far as is known, the initializer never calls this entry with dflag nonzero.

The new_proc entry is called through hcs_$tty_new_proc by the "owning" process in order to change the "using" process. If the channel is not dialed, nothing is done; otherwise, the value of the second argument (nproc) replaces the old contents of wtcb.uproc. This entry is called by the answering service after it is woken up by a user's new_proc command.

Ascertaining the State of a Channel

The tty_state entry is called through hcs_$tty_state in order to find out if a channel is dialed, listening, or neither. It returns the appropriate code based on the values of flags in the WTCB.

Aborting Input and/or Output

The tty_abort entry is called through hcs_$tty_abort as a result of a "resetread" or "resetwrite" control operation. The second argument indicates whether input or output is to be aborted; 1 means input, 2 means output, 3 means both. The general procedure is the same in either case: the input or output chain is freed and the head and tail pointers in the WTCB are zeroed. The "abort" order is forwarded through channel_manager$control to cause any pending input or output being held at higher levels of multiplexing or in the FNP itself to be discarded.

Control Operations

The rest of tty_index--which is to say about two-thirds of it--is devoted to the implementation of the various control operations available through the typewriter DIM. The tty_order entry, called through hcs_$tty_order, implements all the control operations available to users through iox_ and tty_ (except for resetread, resetwrite, and abort, which are implemented by tty_abort, above) as well as the modes operation. In general, these operations have to be forwarded to the major channel through channel_manager$control, in case action by the multiplexer module is required to implement the operation.

The effect of the control operations implemented in tty_index is described in the description of the tty_ I/O module in the MPM Communications Input/Output, Order No. CC92. Rather

than describe the implementation of each control operation in detail, the rest of this section discusses special actions of a nonobvious nature which must be taken in connection with certain operations.

read_status OPERATION

This operation is used by outer-ring programs that wish to find out the state of the input chain without necessarily going blocked. Since all the characters in an input buffer are not necessarily processed by a single call to tty_read, tty_order must check, if there is exactly one buffer in the read chain, whether it contains characters that tty_read has not processed; hence, the buffer tally is compared against wtcb.fchar (which is the offset of the first unprocessed character position in the buffer), and if they are equal, tty_order returns the information that there is no input available in tty_buf. Whenever this result is returned (it is also returned if wtcb.fblock is zero, of course), wtcb.rflag is turned on so that if the caller chooses to go blocked for input later a wakeup is sent when the input arrives.

write_status OPERATION

If this operation reports that output is going on, wtcb.wflag must be turned on so that if the caller goes blocked for output later it can be woken up when the current output chain is sent.

printer_off OPERATION

This operation is intended to prevent typed input from appearing on the terminal, for example when reading passwords. Since echoplex and replay modes both involve input being printed on the terminal by the FNP, special handling is required if an outer-ring program invokes the printer_off operation while the channel is in either of these two modes.

The normal method for turning the printer off is to send the printer-off sequence defined in the special-characters table for the current terminal type as output. This is done by temporarily placing the channel in rawo mode, calling tty_write$locked so that tty_write does not attempt to lock the LCTE (since tty_index has already done so), and restoring the previous setting of rawo mode. If the special-characters table for the current terminal type does not contain a printer-off sequence, no output is sent and a status code of error_table_$action_not_performed is returned.

If the channel is in echoplex mode, disabling the terminal's local-copy function would be ineffective (and has probably already been done). In this case, tty_order sends a modes operation to channel_manager to take the channel out of echoplex mode; the mode bits in the TCB, however, are not changed so that echoplex is automatically restored if a printer_on control operation is requested later.

If the channel is in replay mode, tty_order must tell the FNP to take it out of replay mode when turning the printer off in response to an outer-ring request. As with echoplex mode, when the printer_off operation does take the channel out of replay mode, it leaves tcb.replay on so that replay is restored by a subsequent printer_on operation.

The actual output operation to turn the printer off is effected by an internal procedure, which is also called to disable local copying when the channel is put in echoplex mode (see discussion of "Modes", below).

printer_on OPERATION

The printer_on operation is used to undo the effect of a previous printer_off operation. The same considerations apply: if the channel is in echoplex mode, rather than enabling the local-copy function, tty_order tells the FNP to start echoplexing; if the channel is in replay mode, the FNP is told to resume replaying. For more information, see the discussion of echoplex mode under "Modes," below.

set_terminal_data OPERATION

The set_terminal_data operation is used by tty_ to implement the set_term_type order. It extracts information from the terminal type table (TTT) entry for the requested terminal type, and stores it in the structure defined in the include file terminal_type_data.incl.pl1; this structure is passed to tty_order with the set_terminal_data operation. It includes pointers to the default translation, conversion, special, and delay tables for the terminal type; these tables are copied into tty_tables by calls to tty_tables_mgr, and pointers to the copies are set in the TCB. The default table pointers in the TCB are set to -1 to indicate that the default tables are in effect.

The setting of a terminal type by tty_ may include, besides passing the set_terminal_data operation to tty_order, passing in the initial modes for the terminal type and writing the initial string to the terminal. These operations are accomplished by separate calls to tty_order and tty_write, respectively.

wru OPERATION

The wru ("who-are-you") operation, used to initiate a read of the terminal's answerback, is available only to the owning process. The wtcb.wru flag is turned on so that a wakeup is sent to the requesting process when the FNP returns the answerback sequence even if the sequence does not contain a break character.

MODES

The modes operation is implemented in four phases: first, the caller-supplied character string is validated, and bit strings are set up indicating which modes are to be turned on and off; second, the modes are forwarded to the parent multiplexer, in case any action at the next level of multiplexing is required; third, the appropriate mode bits in the TCB are turned on and off, and any other special actions required for each particular mode are taken; finally, the previous settings of tcb.modes are converted into a character string to be returned to the caller. If any error is discovered during the first phase, no modes are changed, and an error code is returned. The modes operation is implemented in a separated module named tty_modes, which is called by tty_order.

Validation consists primarily of ensuring that all the mode names supplied by the caller are actually names of modes recognized by Multics Communication System. This includes calling channel_manager$check_modes to see if any of the modes are recognized by the multiplexer module. In a few cases, however, further checking must be done. For line length and page length, the mode name ("ll" or "pl") must be followed by an integer that is either zero or in the range of 5<=n<=255. For "fulldpx" (full duplex) or any of the echoing modes (crecho, lfecho, tabecho, or echoplex) the channel must be capable of operating in full-duplex mode; this is determined by looking up the line type (wtcb.line_type) in a table of line types capable of running in full duplex. (This test is performed by fnp_multiplexer$check_modes.) In addition, for crecho, lfecho, and echoplex modes, the FNP has to supply padding when echoing carriage motion characters; tty_modes, therefore, forwards the channel's current delay table to the FNP.

The actual setting of a mode by tty_modes entails turning on or off the corresponding bit in tcb.modes. For most modes, this is all that has to be done; these are modes that affect the operation of the hardcore portion of Multics Communication System only, and not the FNP. For other modes, it is necessary to send an alter_parameters operation to the FNP (see Appendix A for a description of the alter_parameters operation) telling it to turn the specified mode on or off. This is all that needs to be done for tabecho, hndlquit, replay, and polite modes. Echoplex mode is more complicated: when turning it on, not only is the delay

table forwarded, but a printer_off operation is requested to disable local-copy (see the discussion of the printer_off operation, above); similarly, when turning echoplex off, a printer_on operation is requested to restore local copy.

Line length and page length are not recorded in tcb.modes at all, but are reflected in tcb.colmax and tcb.linemax respectively. A zero value for either of these lengths means that checking for maximum line or page length is suppressed.

Privileged Operations

Privileged operations may be defined for a multiplexer; these are invoked through the gates phcs_$tty_control and hphcs_$tty_control and forwarded through priv_channel_manager$priv_control and priv_channel_manager$hpriv_control, respectively. The ones described below are those defined for an FNP channel; they are inplemented by the fnp_multiplexer module. The dump_fnp operation is implemented by the priv_control entry; the others are implemented by the hpriv_control entry.

dump_fnp Operation

In order to obtain the contents of a specified portion of FNP memory, it is necessary to supply a wired buffer for the FNP to write into; accordingly, a sufficient amount of space is allocated in tty_buf. The fnp_info structure contains a lock used to prevent more than one dump_fnp or patch_fnp operation from using the facility simultaneously and a dump_patch_in_progress flag to indicate that such an operation is currently going on. After fnp_multiplexer calls dn355$send_wcd to tell the FNP to dump the specified memory locations into the wired buffer, it waits (using the wait/notify mechanism of pxss) until the DIA I/O is complete. When dn355 receives an interrupt for the submailbox used for the dump_fnp operation (see Section 4 for a more detailed description of this mechanism) it sends a notify for the event reserved for use by Multics Communication System and turns off the dump_patch_in_progress flag. When fnp_multiplexer receives the notify and sees that the flag is off, it copies the data from tty_buf into the buffer supplied by the caller, frees the wired buffer, and unlocks the lock.

patch_fnp Operation

This operation allocates space in tty_buf and uses the "dump_patch_lock" in fnp_info in the same manner as the dump_fnp operation, aside from the essential and obvious difference that caller-supplied data is placed in the wired buffer by fnp_multiplexer and sent to the FNP. The wait/notify mechanism is also used in the same way.

The patch_fnp operation assumes that the caller has previously done a dump_fnp operation, the results of which must be supplied with the patch_fnp call so that the old and new contents of each word of FNP memory being patched can be reported on the operator console and in the syserr log.

fnp_break Operation

This operation is used to manage breakpoints in FNP control tables. (See the discussion of control tables in Section 12 and the description of the debug_fnp command in Appendix B.) The data structure includes an FNP address, an optional channel name, and the action to be performed by the FNP; this action may be to set a breakpoint, to restart a channel stopped at a breakpoint, or to reset a breakpoint. If no channel name is specified, the action is taken to apply to all subchannels of that FNP. The contents of the structure are passed (with appropriate modifications) to dn355$send_global_wcd, which passes them to the FNP in a submailbox with an operation code of "fnp_break".

enable_breakall_mode Operation

This operation must be performed before breakall mode can be used on subchannels of the FNP. A channel name is passed with the control operation; it may be either the name of the FNP channel itself or the name of one of its subchannels. If it is the name of the FNP channel, the mode is enabled for all channels on that FNP, and a flag is set in the fnp_info structure. If the name specified is that of a subchannel, the mode is enabled for that subchannel only, and a flag is turned on in its PCB. For breakall mode to be turned on for a channel, either its PCB flag or the FNP's global flag must be on.

disable_breakall_mode Operation

This operation undoes the effect of an earlier enable_breakall_mode operation by turning off the specified flag. Note that if an FNP channel name is specified, no PCB flags are turned off; thus a disable_breakall_mode operation for an entire FNP does not affect subchannels for which breakall mode has been enabled explicitly.

SECTION 6

HARDCORE UTILITIES

This section describes the operation of two utilities widely used by ring 0 Multics Communication System, tty_lock and tty_space_man; and two special-purpose assembler language subroutines, tty_util_ and dn355_util.

LOCKING AND QUEUING

The program that manages the "channel lock" in each LCTE, used to prevent access to either the LCTE or any of the associated channel's databases when their state is potentially inconsistent, is tty_lock. The following entries are provided: lock_channel and unlock_channel, which lock and unlock LCTEs at call time; lock_channel_int and unlock_channel_int, which lock and unlock LCTEs at interrupt time; flush_queue, which is used to clean out pending queue entries when a channel is terminated; and verify, which is called by verify_lock when a crawlout occurs, to make sure no process leaves ring 0 with a processor lock locked. Because some of these entries are called at interrupt time, tty_lock must be wired.

The lock_channel entry is called by channel_manager, tty_read, tty_write, and tty_index before any references to an LCTE. (dn355 does its own LCTE locking, as described in Section 4.) It uses a stac instruction to attempt to lock the LCTE; if the lock is already locked to some other process, the stac fails, and a wait/notify mechanism is used to determine when the lock becomes unlocked. The notify_reqd flag in the LCTE is turned on; lock_channel calls pxss$addevent to establish an event associated with the lock, tries once more to lock the lock (in case the other process has unlocked it in the interim), and, if it still fails, calls pxss$wait. The process now waits until a notify is sent for the associated event; when this happens, lock_channel tries again to lock the lock, and, if it still cannot do so, reestablishes the event and waits again.

The lock_channel_int entry is called by channel_manager$interrupt. Since pxss$wait must not be called at

interrupt time, this entry does not wait for the lock if it is already locked; instead, it adds an entry to the channel's delay queue. This queue entry contains the interrupt type and associated data. Any queue entries added in this way are processed when the channel is unlocked, as described later in this section.

If lock_channel_int succeeds in locking the channel, it sets a locked_for_interrupt flag in the LCTE. The reason for this is that the interrupt handler may call one of the other entries in channel_manager in order to pass information back to the multiplexer as a result of the interrupt. In this case, lock_channel would find the channel already locked; the setting of the locked_for_interrupt flag would indicate that it was all right for the call to proceed without waiting for the lock to be unlocked, since the operation channel_manager was called to perform is part of the same transaction that locked the channel in the first place.

The unlock_channel and unlock_channel_int entries are almost exactly alike, except for their treatment of the locked_for_interrupt flag. The unlock_channel_int entry turns the flag off before unlocking the channel; the unlock_channel entry does not unlock the channel at all if the flag is on, since the unlock_channel_int entry is due to be called later.

Before unlocking the channel, either unlock entry processes the channel's delay queue: for each entry in the queue, it calls channel_manager$queued_interrupt (which differs from channel_manager$interrupt only in that it does not attempt to lock and unlock the channel) and frees the queue entry. When no more entries remain in the queue, the lock is unlocked.

A separate lock known as the queue lock is used to protect the delay queues from simultaneous modification. This lock is also used to protect changes in the state of a channel lock that could require referencing a queue; i.e., lock_channel_int never locks a channel lock, and no one ever unlocks one, without first locking the queue lock. This strategy is modeled on that used for the coreadd queue lock, which is described in detail in the <u>Multics Storage System PLM, Order No. AN61</u>.

Once a channel lock is unlocked, if notify_reqd is on, either unlock entry calls pxss$notify so that the process that is waiting in lock_channel can try once again to lock the lock.

The verify entry is called by verify_lock at crawlout time. It checks to see if the global tty_buf lock, tty_buf.slock, is locked to the current process; if it is, cleanup_locks crashes

the system, since the state of tty_buf itself must be assumed to be inconsistent. Similarly, the system crashes if queue lock is locked to the current process, or any channel lock for a channel whose "special lock" flag is on, indicating that the channel's multiplexer (rather than tty_lock) manages the lock.

SPACE MANAGEMENT

Wired buffer space in tty_buf is managed by entries in tty_space_man. While manipulating the buffer pool, these entries must have the global or "system" lock, tty_buf.slock, locked; to avoid losing the processor while this lock is locked, they must run wired and masked. Three kinds of action are performed by tty_space_man: allocation, freeing, and the setting of "space_needed" flags. The formats of the blocks manipulated by tty_space_man are described in Section 2.

Allocation

Space in tty_buf may be allocated either for use in buffer chains (for input and output data) or for any of the various control structures described in Section 2. Buffers are allocated by the entries get_buffer and get_chain; other control blocks are allocated by the entry get_space. The principal difference is that the size of a buffer is always a multiple of 16 words, with a maximum size of 128 words; get_space, on the other hand, can allocate space in any even number of words. The buffer allocation entries set the size code in each allocated buffer (the meaning of the size code is explained in the description of data buffers in Section 2).

The get_buffer entry is called to allocate a single buffer whose size is a multiple of 16 words; get_chain is called to allocate a chain of such buffers, all of equal size, each containing a relative pointer to the next one in the chain; get_space allocates a block of arbitrary size. In any case, tty_space_man searches the chain of free blocks from the lowest address, looking for the smallest free block that is large enough to contain a buffer or block of the requested size. If a chain is being allocated, this process is repeated for each requested buffer, and each buffer allocated (except the last one) has its "next" pointer set to the offset of the following buffer. The buffer allocation entries also update a field in the LCTE of the channel on whose behalf they were called, to indicate how much input and output space is currently assigned to that channel. This information is used by dn355 and tty_write to determine how much input and output, respectively, to accept for the channel. The contents of a buffer or block are set to zero upon allocation, except for the size code and forward pointer if appropriate. If tty_space_man is unable to find a block or blocks of the requested size, it returns a null pointer; it is the responsibility of the caller to take appropriate action.

Freeing

The free_buffer entry is called to free a single buffer; free_chain is called to free a chain of buffers, and return the number of buffers in the freed chain. The free_space entry is called to free a block of arbitrary size. A block allocated by get_space must be freed by free_space; buffers allocated by get_buffer or get_chain must be freed by free_buffer or free_chain, although buffers that were allocated separately can be freed as a chain, and vice versa. The free_buffer and free_chain entries deduce the size of each buffer being freed from its size code; free_space must be told the size of the block to be freed. The last buffer in a chain to be freed by free_chain is identified by a forward pointer of 0.

Each buffer or block freed is threaded into the free chain at the appropriate point (the free chain is sorted by increasing address). If the newly free block is immediately preceded and/or followed by an already free block, the adjacent blocks are combined into a single, larger free block.

Once the supplied buffers have been freed, the freeing entries process any pending "space_needed" requests, as explained below.

Needed Space

When tty_write is unable to allocate any buffers in which to build an output chain, it calls tty_space_man$needs_space to ensure that a wakeup is sent to the calling process when more buffers become available. This entry sets the "space_needed" flag in the channel's LCTE, and also sets a flag in the tty_buf header to indicate that there is an LCTE with its space_needed flag set.

When tty_space_man finishes freeing a buffer, chain, or block, it checks the flag in the tty_buf header and, if it is on, searches the entire LCT for entries with the space_needed flag on. For each one it finds, it calls channel_manager$interrupt with an interrupt type of "space_available"; tty_interrupt handles this interrupt by sending a wakeup to the process using the channel. This enables each process to retry its call to tty_write; of course, there is no guarantee that it will now be possible to allocate buffers for the channel, but if not, tty_space_man$needs_space is called again. When tty_space_man is finished searching the LCT, it checks again to see if any space_needed flags have been turned on again as a result of any of the interrupts it generated. If not, it turns off the flag in the tty_buf header.

tty_util_ Module

The tty_util_ module contains entries called by tty_read and tty_write to scan and/or translate strings of input or output characters using EIS instructions. The occasions on which these entries are called are described in Section 5; a brief summary of the available entries is included here.

mvt
: translates a string under control of a translation table.

scm
: searches a string for a character with its 400(8) bit or its 200(8) bit on, depending on how it is called.

tct
: searches a string for a character whose indicator in a conversion table is nonzero, and returns the position of the character and the value of the indicator.

find_char
: like tct, except that it also checks explicitly for characters with either of their two high-order bits on, and for white space combinations beginning with a blank.

illegal_char
: scans a string for a character with either or both of its two high-order bits on.

All of these entries are provided with a pointer to, and the length of, the string to be processed. All except mvt update the pointer and the length to reflect how much of the string was scanned before the specified condition was met.

dn355_util Subroutine

The dn355_util subroutine contains a single entry, dn355_util$compute_parity, which is used to ensure that the PCW sent to the DIA for interrupting, bootloading, or dumping the FNP has odd parity. If the parity of the PCW is not already odd, the subroutine makes it so by turning on bit 22.

SECTION 7

INITIALIZATION OF HARDCORE DATA BASES

The dynamic data bases used by ring 0 Multics Communication System, i.e., dn355_mailbox, dn355_data, tty_area, tty_tables, and tty_buf, are initialized in several stages: preliminary initialization during hardcore initialization (collection 1); LCT initialization during answering service initialization (after a startup, multics, or go command); and the initialization of per-channel and per-multiplexer data bases as the multiplexers are initialized and loaded. These data bases are all described in Section 2.

PRELIMINARY INITIALIZATION

The program called initializer, which runs during collection 1 initialization, calls fnp_init to process configuration cards relating to Multics Communication System. (See the Multics Operators' Handbook, Order No. AM81, for a description of these cards.) First it calls get_main to allocate wired storage for tty_buf, using the size specified on a PARM TTYB card; if no such card is present, a default size of 5120 words (5K) is used. The absolute address of the base of tty_buf is calculated and stored. The size of the circular buffer is determined from a PARM TTYQ card; if no such card is present, a default size of 256 words is used. The necessary amount of space for the circular buffer is reserved at the end of the tty_buf header; the remainder of tty_buf, starting at the next 0 mod 16 address, is marked as the free pool, which initially consists of one free block. Then the areas in tty_tables and tty_area are initialized as empty areas.

Finally, dn355_data and dn355_mailbox are initialized in accordance with all FNP cards found in the configuration deck. These cards are checked for validity and consistency; for each one, the IOM number and IOM channel assigned to the specified FNP are copied into the entry in dn355_data for that FNP, and the mailbox pointer is set to address the correct mailbox area. Calls are made to iom_manager to assign dn355$interrupt as the interrupt handler for the specified IOM channels; in addition,

fnp_init calls iom_manager$iom_set_list for each FNP, since iom_manager requires this call in order to perform I/O properly.

LCT INITIALIZATION

During answering service initialization, the initializer process scans the CDT to determine how many communications channels are defined; based on this, and on the value specified for the spare_channel_count keyword, it determines the number of LCTEs required, and calls the gate entry hphcs_$lct_init. This call is forwarded to priv_channel_manager$lct_init, which calls tty_space_man$get_space to allocate space for the LCT and also allocates space for the LCNT in tty_area. It then assigns LCTEs at the beginning of the LCT to however many FNPs are configured, and sets their special_lock flags to indicate that channel_manager is not to attempt to lock these channels.

MULTIPLEXER INITIALIZATION

Before loading a multiplexer, the answering service calls hphcs_$init_multiplexer, which call is forwarded through priv_channel_manager to the initialization entry of the appropriate multiplexer module, as explained in Section 3. In the case of an FNP, this entry is fnp_multiplexer$init_multiplexer. It initializes the specified FNP's fnp_info structure in dn355_data, and allocates a contiguous block of space for PCBs for all the subchannels of the FNP. The array of channel names passed to this entry is sorted in ascending order, so init_multiplexer knows that the channels for any given line adapter (HSLA or LSLA) are contiguous; this enables it to assign them contiguous PCBs and store the PCB index of the first channel on each adapter in the fnp_info structure, thereby reducing the number of entries dn355 has to scan in order to find the PCB for a channel specified in a submailbox from the FNP.

CHANNEL INITIALIZATION

Before issuing a "listen" control operation to a nonmultiplexed channel, the answering service calls hphcs_$init_channel; this call is forwarded through priv_channel_manager to tty_index$init_channel, which allocates the channel's WTCB and TCB as explained in Section 5.

SECTION 8

TOOLS AND DEBUGGING AIDS

A few commands are provided to allow a user process to obtain information about the current state of the CS portion of Multics Communication System and its associated databases; they can be helpful in debugging Multics Communication System. This section describes these commands. It also includes information on error messages that Multics Communication System may write on the syserr console and/or into the syserr log, as well as some pointers on analyzing Multics Communication System-related system crashes.

TOOLS

Three commands are discussed in this section: tty_meters, which displays the metering information kept in tty_buf by Multics Communication System (see Section 2 for descriptions of the individual structure items); tty_dump, which displays the current state of a single channel; and tty_analyze, which displays information extracted from a system dump. Command descriptions of some of these commands appear in Appendix B; others appear in the Multics Administrators' Manual (MAM) -- Communications, Order No. CC75.

tty_meters

The tty_meters command uses information stored in the header of tty_buf to derive statistics on the behavior of Multics Communication System and of the various communications channels. It uses two temporary segments to keep two copies of tty_buf, and one for a copy of dn355_data. The two copies of tty_buf can be thought of as the "current" copy and the "old" copy. The current copy is filled in from ring 0 every time the command is invoked; the old copy, initially all 0, is updated from the current copy whenever the -reset control argument is used. The statistics printed by the command reflect the differences between the values in the old copy and in the current copy. The command description for tty_meters is in MAM -- Communications, Order No. CC75.

The command can operate in either normal mode or long mode. (It operates in normal mode unless the -long control argument is used.) In normal mode, only the header of tty_buf is copied from ring 0, up to but not including the first word of the circular buffer. In long mode, all of tty_buf is copied, and additional information on how many terminals of each type are dialed up is derived by scanning all the WTCBs. The values of wtcb.send_output, wtcb.rflag, and wtcb.wflag are also used in long mode to determine how many channels are currently sending input or receiving output. Because the length of the header is specified in ring_zero_meter_limits_ASCII_, use of long mode requires access to the phcs_ gate.

The copy of dn355_data is used to extract the version identifier of the Multics Communication System running in each configured FNP. This information is displayed at the beginning of the output of tty_meter, along with the FNP core image name specified in the CDT. The average number of free buffers in each FNP is also derived from dn355_data.

tty_dump

The tty_dump command displays various databases of a specified communications channel. It makes copies of tty_area, tty_buf, and dn355_data. It can be used to display information derived from either a running system or an FDUMP. Use of this command with a live system requires access to the phcs_ gate. The channel can be selected by specifying either the channel name or a person name. If a channel name is specified, the LCNT is searched for that channel's entry. If a person name is specified, the answer table is searched for a logged-in user having that name; if one is found, information is displayed about the communications channel(s) assigned to that user's process by the answering service. The command description for tty_dump is in MAM -- Communications, Order No. CC75.

The format of the information displayed by tty_dump depends on the multiplexer type of the specified channel. For a nonmultiplexed channel, the WTCB and TCB are displayed, along with the contents of any current input and/or output chains. The -subchannel control argument may be used to indicate that the database of the parent multiplexer, or at least that portion of it that concerns the specified channel, is to be displayed (for a physical channel of an FNP, this means to display the channel's PCB). The -all control argument specifies that the databases associated with the channel at all levels of multiplexing (up to and including the physical FNP channel) are to be displayed. The -lcte control argument indicates the LCTE of the specified channel (and its parents if -all is specified) is to be displayed.

For channel types other than nonmultiplexed ("tty") channels, a subroutine that displays the associated multiplexer databases must be provided. Its name must be of the form TYPE_dump_, where TYPE is the name of the multiplexer type (e.g., user1_dump_ for the multiplexer type "user1"). The calling sequence of this subroutine is described below, using a multiplexer type of "user1" for the sake of example.

```
declare user1_dump_ entry (pointer, pointer, pointer, fixed
     bin, bit (1));

call user1_dump_ (ttybp, areap, database_ptr, subchan,
     brief_sw);
```

where:

ttybp (Input)
: is a pointer to the base of tty_buf.

areap (Input)
: is a pointer to the base of tty_area.

database_ptr (Input)
: is a pointer to the multiplexer's database.

subchan (Input)
: is the subchannel number of the channel about which information was requested. A subchannel number of -1 indicates that information is requested for all subchannels.

brief_sw (Input)
: is "1"b if the -brief control argument to tty_dump was specified; otherwise it is "0"b.

A system-supplied example of such a subroutine is vip7760_dump_.

An entry to tty_dump called print_chain is provided to allow multiplexer dumping subroutines to share the code in tty_dump that displays the contents of an input or output chain. Its calling sequence is described below.

```
declare tty_dump$print_chain entry (pointer, char(*), fixed
     bin, bit (1));

call tty_dump$print_chain (ttybp, chain_name, chain_start,
     brief_sw);
```

where:

ttybp (Input)
: is a pointer to the base of tty_buf.

chain_name (Input)
: is a character string identifying the type of chain; it is printed before the contents of the chain.

chain_start (Input)
: is the offset in tty_buf of the first buffer in the chain.

brief_sw (Input)
: is "1"b if only the offset, size, and flags associated with each buffer are to be displayed. If it is "0"b, the contents of each buffer are displayed as well.

tty_analyze

The tty_analyze command produces formatted output describing the contents of tty_buf taken from a Multics segment dump. The command description for tty_analyze appears in Appendix B. It calls the extract command to copy tty_buf, tty_area, dn355_mailbox, and dn355_data from a specified dump in >dumps. It then loops through the array of PCBs for each configured FNP; for each one, it displays the contents of the PCB. If the channel is not multiplexed, it then displays the contents of the WTCB; otherwise it calls a subroutine (described below) to display the multiplexer database, and searches the LCNT for all subchannels of the multiplexed channel. This process is repeated through all levels of multiplexing (down to the WTCB level) until the databases of all descendants of each physical channel have been displayed. For each channel, the delay queue, if any, pointed to by the LCTE is also displayed. In addition, for each channel having a read and/or write chain it prints the addresses of all buffers in each such chain. If the -long control argument is specified, the contents of each buffer are also printed (in octal). Each buffer (and each database) thus accounted for is marked with a special pattern so that multiple uses of a single block can be detected.

For each multiplexer type (other than "mcs" and "tty"), a subroutine must be supplied that displays the contents of multiplexer databases. Its name must be of the form TYPE_analyze_, where TYPE is the name of the multiplexer type (e.g., user1_analyze_ for the multiplexer type "user1"). The calling sequence of this subroutine is described below, using a multiplexer type of "user1" for the sake of example.

```
declare user1_analyze_ entry (pointer, pointer, fixed bin,
     entry, bit(1));

call user1_analyze_ (ttybp, areap, devx, check_used_entry,
     longsw);
```

where:

ttybp (Input)
: is a pointer to the base of tty_buf.

areap (Input)
: is a pointer to the base of tty_area.

devx (Input)
: is the device index of the multiplexer channel.

check_used_entry (Input)
: is the entry to be called to mark a buffer or control block as having been accounted for; the calling sequence is described below.

longsw (Input)
: is "1"b if the "-long" control argument to tty_analyze was specified; otherwise it is "0"b.

A system-supplied example of such a subroutine is vip7760_analyze_.

For every control block or buffer that it displays, the subroutine should call the check_used_entry passed to it by tty_analyze to account for the space. The calling sequence of this entry is as follows:

```
declare check_used_entry entry (pointer, fixed bin)
     variable;

call check_used_entry (block_ptr, nwords);
```

where:

block_ptr (Input)
: is a pointer to the buffer or block to be accounted for.

nwords (Input)
: is the length of the buffer or block in words.

After all channels have been checked, tty_analyze displays the addresses of all FNP delay queue entries created by dn355 (see Section 4). Then it searches dn355_mailbox for any currently-active submailboxes that contain an I/O command of WTX, and displays the addresses of any pseudo-DCW lists pointed to by such mailboxes. If the -long control argument is specified, the contents of these blocks are also displayed. Then tty_analyze follows the free block chain starting at tty_buf.free, and any invalid chain pointers are reported. Finally, a list is printed

of the addresses of all blocks in tty_buf not otherwise accounted for.

SYSERR MESSAGES

The messages discussed here are those messages generated by calls to syserr without crashing Multics. (Crashes generated by Multics Communication System are discussed later in this section.) Almost all nonfatal error messages produced by Multics Communication System are the result of conditions detected by the FNP.

FNP Crashes

When an FNP crashes (as described in Sections 13 and 16) it sends an "emergency" interrupt to the CS; dn355 handles this interrupt by interpreting the crash data in words 6 and 7 of the mailbox header of that FNP (mailbox headers are described in Sections 2 and 16). Using the type of FNP fault and the contents of the FNP instruction counter, it looks up the appropriate message in dn355_messages, and prints it on the syserr console. It then signals the process that bootloaded the FNP (which is generally the initializer process), and reports hangup conditions on all channels connected to that FNP.

Other FNP Messages

When a submailbox from the FNP contains an "error message" operation code, the error code included in the command data is used to index an array of messages in dn355_messages to find the one to print on the console. These messages describe nonfatal FNP errors, such as certain types of DIA I/O error, possibly runaway HSLA subchannels, and the like. They do not normally require any action, but they are sometimes useful in tracking down other problems. If some FNP channels do not seem to be behaving properly, or if an FNP crash is not readily analyzable, any error messages sent by that FNP may provide useful additional information. These messages all indicate probable hardware problems.

CRASHES GENERATED BY MULTICS COMMUNICATION SYSTEM

The CS portion of Multics Communication System elects to crash (by calling syserr) if certain inconsistent or "impossible" conditions are detected; for example, an attempt to unlock a lock that turns out to have been locked by some other process, an unrecognizable submailbox sent from the FNP, or an interrupt on an unrecognized level. These conditions are all quite rare, and the associated syserr messages are generally self-explanatory. It is often a good idea to dump the FNP as well as the CS when an Multics Communication System-related crash occurs, particularly

one resulting from a faulty submailbox. There is a class of errors detected by tty_space_man (see Section 6) indicating that the buffer pool is in an inconsistent state; tty_analyze may be useful here, as is visual examination of a dump of tty_buf. In general, dn355_mailbox, dn355_data, and tty_buf should all be dumped after an Multics Communication System-related or Multics Communication System-suspected crash.

Two particular types of crash message are insufficiently explicit to be self-explanatory; both are accompanied by codes that can be used to determine the nature of the error. These codes are discussed below.

Lock Errors

A message of the form:

dn355: lock ^= processid, error = n

indicates that dn355 attempted to unlock a lock but discovered that the lock did not contain the ID of the currently running process. The code n can have either one of the following values:

5 -- the circular buffer lock was invalid

6 -- dn355$interrupt found the FNP channel lock invalid

Free Space Errors

A message of the form:

tty_space_man: error of the Nth kind: ERR_TYPE

indicates an inconsistency or other problem in the buffer pool. These conditions are detected by tty_space_man (see Section 6.)

N and ERR_TYPE can have the following values:

N	ERR-TYPE	Explanation
1	Mylock error	A process attempting to lock the tty_buf lock already had it locked
2	Unlock error	A process attempting to unlock the tty_buf lock did not have it locked
3	Bad address	One of the freeing entries was called with an address outside the free pool

4	Already free	An attempt was made to free space that was already free

SECTION 9

OVERVIEW OF THE FNP SOFTWARE

RESPONSIBILITIES OF THE FNP

The FNP-resident portion of Multics Communication System has the following major functions:

1. controlling the various communications channels in accordance with the appropriate communications protocols;

2. keeping the CS informed of the current state of each channel;

3. presenting input data from the various channels to the CS;

4. transmitting output supplied by the CS to the appropriate channels;

5. performing all echoing functions supported by Multics Communication System (linefeed and carriage return echo, tabs echoed as spaces, echoplex, replay).

STRUCTURE OF FNP MULTICS COMMUNICATION SYSTEM

The major components of the FNP software are listed below:

1. Scheduler - handles interrupts, decides which routine to run, manages timers. The scheduler is discussed in greater detail in Section 11.

2. Terminal control functions: control tables and the control table interpreter - implement individual communications protocols, keep track of the state of each channel. These functions are discussed in greater detail in Section 12.

3. Hardware managers - control the direct interface adapter (DIA), low-speed line adapter (LSLA), high-speed line adapter (HSLA), and the FNP console. Hardware managers are discussed in Section 13.

4. Utilities - buffer space management, fault handling, tracing functions, etc. Utilities are discussed in Section 14.

Each of these components is implemented by one or more modules written in the FNP assembler language, 355MAP. Each module is assembled separately using the map355 command, which invokes the assembler (in an ordinary Multics process) under GCOS simulation; the assembled modules are combined into a "core image" for the FNP by the bind_fnp command. The resulting core image is loaded into the FNP by the Multics initializer at system initialization time, or at any time in response to the operator command load_mpx. See MAM-Communications, Order No. CC75 for descriptions of the map355 and bind_fnp commands, and Section 15 for a description of the bootloading of the FNP.

The modules that compose the core image are listed below. For each module, the long or "external" form of the name is the one used in specifying the source and object segments for the module, and is the name used in this document to refer to the module; the short or "internal" form is the symbol that defines the module within the core image and is stored in the module itself as part of the "module chain" used in printing dumps of the FNP (see Section 16).

External name	Internal name	Comments
scheduler	sked	
interpreter	intp	control table interpreter
control_tables	ctrl	main control tables module
other control tables modules	----	see Section 12
dia_man	dia	DIA manager
lsla_man	lsla	LSLA manager
hsla_man	hsla	HSLA manager
console_man	cons	FNP console manager
utilities	util	

ic_sampler	icsamp	meters values of the instruction counter at interrupt time (see Section 11)
trace	trac	performs memory tracing functions (see Sections 14 and 16)
breakpoint_man	bkpt	manages breakpoints in control tables (see the description of the debug_fnp command in Appendix B)
init	init	FNP initialization program

Data Bases

A variety of common databases are used by all these modules to communicate information about the states of the various channels and hardware adapters; the data bases are described in Section 10. The most important of these is the terminal information block (TIB); there is one TIB for every configured channel, which describes the current state of the channel. The TIB is referenced by almost every module in the FNP; as a matter of software convention, the address of the TIB for the channel of current interest is held in index register 1.

CHANNEL MANAGEMENT

It may be convenient to view each channel as being managed by its own "process" in the FNP, where the state of the "process" is described by its TIB. The control tables are run in such a "process" whenever there is any work to do for the associated channel; when the "process" is not running, a field in the TIB (t.cur) identifies the place in the control tables at which execution last stopped.

INTERRUPTS AND SCHEDULING

The FNP is driven by interrupts, which come primarily from four sources:

1. the DIA, either sent from the CS or used to signal the completion of DIA I/O;

2. the LSLA reporting status resulting from the processing of an input or output frame;

3. the HSLA reporting status for a particular subchannel;

4. the interval timer, when a software-established timer runs out.

A routine that handles an interrupt does very little other than identifying the reason for the interrupt and scheduling another routine to do the work required by the event that caused the interrupt; for example, the control tables are never run at interrupt time, but may be either scheduled to run as a result of a timer interrupt or called by another scheduled routine. The interrupt and scheduling mechanisms are described in Section 11.

DATA PATHS

Input

When a user types a message-ending character (typically newline) an interrupt is generated (on an HSLA channel) or the character is recognized in an LSLA input frame, and "break character" status is sent to the control tables for the channel; the address of the beginning of the input message is stored in the TIB. The control tables, possibly after examining the data, execute a "sendin" operation block, which queues up a request for DIA I/O. When dia_man runs, it fills in a submailbox either with the input itself or with an operation code saying that input is available and a tally of characters in the message. In the latter case, the CS replies with another submailbox telling the FNP where to put the data; dia_man builds a DCW list to copy the data from FNP buffers into CS main memory, and when the I/O completes it frees the buffers in the FNP.

Output

When output is ready to be sent from the CS, the CS prepares a mailbox telling the FNP the location of a block of "pseudo-DCWs" containing the addresses and tallies of output buffers in the CS. After reading this submailbox, dia_man reads in the pseudo-DCWs, and uses the information in them to read the data from CS memory into buffers in the FNP. The address of the first of these buffers is stored in the TIB, and dia_man calls the interpreter to start the control tables. The latter generate a call to lsla_man or hsla_man to send the data characters to the appropriate channel; when the last character in a buffer is sent to the channel, the buffer is freed.

SECTION 10

FNP DATA BASES

This section describes various data bases used by the FNP software. Some data bases used primarily by the hardware, but of interest to the systems programmer, are described in the Datanet 355 Macro Assembler Program manual, Order No. BB98.

The fields in most of the data bases described in this section are defined by macros that are included in those modules that reference the various data bases. The names given in these descriptions are the names used in the macros. In some cases, the reader is referred to the macro definitions themselves for a detailed list of the contents of the data bases.

SYSTEM COMMUNICATIONS REGION

The system communications region contains a variety of information of general interest to the Multics Communication System software. It begins at absolute location 640 (octal) in FNP memory and currently extends through location 677, but space is reserved for its expansion up to location 775. Its contents are described by the "comreg" macro. Fields in the system communications region have names beginning with ".cr".

IOM TABLE

The IOM table is a 32-word area describing the configuration of the channels on the Input/Output Multiplexer (IOM) of the FNP. It is used during FNP initialization. The address of the IOM table is kept in .criom in the system communications region (see above).

Up to 16 channels may be configured on the IOM, and each entry in the IOM table corresponding to such a channel is two words long. The format of the IOM table is described below.

Words	Meaning
0- 1	1st device element (FNP console)
2- 3	2nd device element (FNP reader)
4- 5	3rd device element (FNP printer)
6- 7	4th device element (not implemented)
10-11	5th device element (DIA)
12-13	6th device element (not implemented)
14-15	7th device element (HSLA 0)
16-17	8th device element (HSLA 1)
20-21	9th device element (HSLA 2)
22-23	10th device element (LSLA 0)
24-25	11th device element (LSLA 1)
26-27	12th device element (LSLA 2)
30-31	13th device element (LSLA 3)
32-33	14th device element (LSLA 4)
34-35	15th device element (LSLA 5)
36-37	16th device element (clock)

Device Element Format:

word 0 - flag word
word 1 - address of HSLA table or LSLA table if
device is HSLA or LSLA respectively

Flag Word Format:

```
  Bit      Description

    0      multiplexed channel (not used)
    1      device released (not used)
    2      asynchronous device (not used)
-------------------------------------------
  3-5      adapter number (if LSLA or HSLA)
-------------------------------------------
    6      T & D in control (not used)
  7-8      character length code (not used)
-------------------------------------------
 9-13      device type code
================================== crosses 3-bit boundary
14-17      device speed code (for LSLA only)
-------------------------------------------
```

Device Type Codes:

Name	Code	Meaning
dnimp	00	not implemented
dclock	01	clock
ddia	02	DIA
dhsla	03	HSLA
dlsla	04	LSLA

dcon	05	console
dprint	06	printer

HARDWARE COMMUNICATIONS REGIONS

There is one 16-word hardware communications region associated with each configured LSLA, and one for each subchannel of each HSLA. The hardware communications region for the first LSLA (which is called LSLA 0) starts at absolute location 500 (octal); the one for LSLA 1 starts at location 520, etc. The hardware communications region for subchannel 0 of HSLA 0 starts at location 1000 (octal); for subchannel 1 of HSLA 0, at 1020, etc. Since there are 32 subchannels per HSLA, the hardware communications regions for each complete HSLA occupy 512 decimal (= 1000 octal) words, so that the hardware communications regions for the second HSLA (if more than one is configured) start at location 2000, and for the third at 3000.

The hardware communications regions contain the indirect control words (ICWs) used by the LSLA and HSLA hardware in sending and receiving data and for storing status. Some of the fields in the hardware communications region are used differently depending on whether the device is an HSLA subchannel or an LSLA, as noted in the description below. The hardware communications region is defined by the "hwcm" macro.

Words	Mnemonic	Meaning
0- 1	h.ric0	primary receive icw
2- 3	h.ric1	alternate receive icw
4- 5	h.sic0	primary send icw
6- 7	h.sic1	alternate send icw
10	h.baw	base address word (HSLA)
	hcmt1	count of missed STX frames (LSLA)
11	h.sfcm	address of software comm region
12-13	h.mask	mask register (HSLA subchannel 0 only on DN355 and DN6600; HSLA Subchannels 0, 8,16, and 24 on DN6670)
12	hcmt2	count of attempts to resync (LSLA)
13	hcmt3	maximum number of successive resync attempts (LSLA)
14-15	h.aicw	active status icw
16-17	h.cnfg	configuration status (HSLA)
16	hcmt4	temporary counter (LSLA)

SOFTWARE COMMUNICATIONS REGIONS

There is a software communications region for each LSLA and for each configured HSLA subchannel. The software communications region contains information about the state of the relevant hardware channel that is primarily of interest to the software

module that manages the relevant adapter (i.e., lsla_man or hsla_man). The address of each software communications region is kept in the corresponding hardware communications region. Fields in the software communications region have names beginning with "sf.". Flags in the software communications region have names beginning with "sff".

LSLA Software Communications Region

Since there cannot be more than six LSLA software communications regions (one for each configured LSLA), space is reserved for as many as will be needed at the end of the module lsla_man, depending on the number of LSLAs specified in the bind file (see the description of bind_fnp command in Section 17). The format of an LSLA software communications region is described by the "sfcm" macro with a parameter of "lsla".

HSLA Software Communications Region

Since there is one software communications region for each configured HSLA subchannel, and therefore anywhere from zero to 32 for each HSLA, space is not reserved for them when the core image is bound; rather, they are allocated as needed during FNP initialization (see Section 15). The format of an HSLA software communications region is described by the "sfcm" macro with a parameter of "hsla".

LSLA TABLE

There is an LSLA table for each configured LSLA, and an entry in the LSLA table for each configured time slot on each LSLA. (See Section 13 for an explanation of LSLA time slots.) The LSLA table entries are used to make the connection between LSLA time slots and actual communications channels.

The starting address of the LSLA table for each LSLA is kept in the IOM table entry for that LSLA. Since time slot 0 is reserved for use as a T & D slot, the first entry of each LSLA table is unused.

The format of an LSLA table entry is described below.

Name	Offset	Meaning
lt.flg	0	flag word (bits 0-14)
lt.sid	0	slot identifier (bits 15-17)
lt.tib	1	TIB address

Flag Word (lt.flg):

Symbol	Bit	Mask	Meaning
ltfbbk	04	020000	line break status received
ltfdcw	05	010000	new dcw list pending for line
ltfst	06	004000	request for status is pending
ltfbrk	07	002000	sending delays for line break
ltfbex	08	001000	break on next input character
ltfrqs	09	000400	current setting of request to send
ltfdtr	10	000200	current setting of data terminal ready
ltfibm	11	000100	IBM-type (odd parity) terminal
ltfesc	12	000040	escape sent before command
ltfste	13	000020	status character expected
ltfskp	14	000010	skip next character

Time Slot Identification Codes (lt.sid):

Name	Code	Meaning
lt10	0	10 cps
lt30a	1	30 cps, slot 1
lt30b	2	30 cps, slot 2
lt30c	3	30 cps, slot 3
	4	invalid
lt15a	5	15 cps, slot 1
lt15b	6	15 cps, slot 2
	7	unused slot

HSLA TABLE

There is an HSLA table for each configured HSLA, and an entry in each HSLA table for each configured subchannel. The HSLA table is used at FNP initialization time; it is the path by which information contained in the CDT for each channel is passed to the FNP.

The starting address of the HSLA table for each HSLA is kept in the IOM table entry for that HSLA. The format of an HSLA table entry is described below, and is defined by the "hslatb" macro.

word 0 - basic subchannel data described below

word 1 - address of TIB

Basic Subchannel Data:

```
htfcc      0         400000   concentrator channel (not used)
htfpl      1         200000   private line modem or hardwired
htfasy     2         100000   asynchronous channel
-----------------------------------
htfop1     3         040000   option 1:
                              automatic baud rate detection (async)
                              EBCDIC code (binary synchronous)
htfop2     4         020000   option 2 (not used)
-----------------------------------
htfmdm     5-8       017000   modem type (4 bits)
===================================
htflnt     9-13      000760   line type (5 bits)
===================================
htfspd     14-17     000017   device speed code (4 bits)
-----------------------------------
```

Modem Types:

1	Bell 103A or 113
2	Bell 201C
3	Bell 202C5
4	Bell 202C6
5	Bell 208A
6	Bell 208B
7	Bell 209A

Line types are described as part of the Terminal Information Block (below).

Device Speed Codes:

Code	Async Speed	Sync Speed
01	75	2000
02	110	2400
03	134.5	3600
04	150	4800
05	300	5400
06	600	7200
07	1050	9600
10	1200	19200
11	1800	40800
12	option	50000

TERMINAL INFORMATION BLOCK

There is a terminal information block (TIB) for each configured channel. Each TIB is allocated at FNP initialization time; it describes the state of the channel from a software point of view, and is referenced by almost every module in the system. The address of the TIB is kept in the corresponding LSLA or HSLA table entry, and also, in the case of HSLA channels, in the corresponding software communications region. Names of fields in the TIB begin with "t."; names of bits in the status word (t.stat) begin with "tsf"; names of flags in TIB flag words begin with "tf". The format of a TIB is described by the "tib" macro.

TIB TABLE

At the base of the init module is a table of 2-word entries, one entry for each configured channel. The first word of each entry contains the address of the channel's TIB; the second word contains the address of a buffer containing a queue of DIA I/O requests on behalf of the channel. If there are no such requests, the second word is zero. This table is used by dia_man to service each channel in turn, as described in Section 13.

BUFFER POOL

The buffer pool consists of free blocks and buffers allocated for various purposes, such as channel I/O, scheduler queues, DIA request queues, etc. The formats of various types of blocks are described below.

Free Block

A free block may be any even size. The first two words are used to identify a free block and thread it into the free chain. The head of the free chain is pointed to by .crnxa in the system communications region (above).

word 0	address of next free block (zero if this is last one)
1	size of free block in words

Input/Output Buffer

Buffers used for input from and output to channels are allocated in 32-word blocks as described in Section 14. Each buffer starts at a 0 mod 32 address. The format of a buffer allocated for channel I/O is shown below, and is defined by the "buffer" macro.

bf.nxt	0	address of next buffer in chain
bf.siz	1	size of buffer (bits 0-2) (in 32-word blocks)
bf.flg	1	flags (bits 3-8)
bf.tly	1	tally of buffer (bits 9-17)
bf.dta	2	start of data

Buffer Flag Word (bf.flg):

bfflst	3	040000	last buffer in message
bffbrk	4	020000	buffer contains break character
bfftra	5	010000	throw away buffer when done inputting
bffrpy	6	004000	last buffer in replay chain
bffctl	7	002000	buffer consists of keybd/prtr control
bffhld	8	001000	hold output buffer until dmpout operation

Echo Buffer

Echo buffers are used by lsla_man and hsla_man to store characters to be sent to a channel in any of the four echoing modes (tab echo, linefeed echo, carriage return echo, and echoplex). They are 32 words long. The format of an echo buffer is described below.

word 0	character address of next place to store an echo character
1	character address of next character to be output
2	bits 0-8: number of outstanding characters to be echoed
remainder	up to 59 data characters

If an attempt is made to add a character to an echo buffer that is full of characters not yet echoed, the TIB flag tfbel is turned on. This indicates to the LSLA or HSLA manager that a BEL character should be output, in order to warn the user that echoed characters have been lost.

DIA Request Queue

The control tables enqueue requests for DIA I/O by means of "signal" and "sendin" operations; these requests are put in buffers for later processing by dia_man (see Section 13). There

is a separate queue for each TIB; the starting address of each queue is kept in the TIB table (see above).

The format of a buffer in a DIA request queue is described below.

word 0	address of next buffer in queue (zero if this one is last)
1	number of queue entries currently in buffer
2	address of next available location in buffer
3-31	queue entries

The first word of a queue entry has the following format:

bits 0-5	flags (only used if operation is "accept input")
6-8	number of data words
9-17	operation code to be sent to CS

If the word count (bits 6-8) is nonzero, one or more words of descriptive data follow, depending on the operation code.

The following flags are defined:

0	400000	request is active
1	200000	request has been rejected by CS
2	100000	there is a "quit" or "hangup" signal later in this queue

Error Message Queue

In a few cases, the FNP software sends error messages to the CS to be printed on the syserr console. These error messages are queued in a single queue built in chained buffers, and processed by dia_man. The queue has the same format as the DIA request queue (above).

Delay Timing Tables

The delay timing tables are used when echoing input characters to determine how many delay characters are to be sent following a carriage motion character. When the CS sends an

"Alter Parameters" operation to enter echoplex, linefeed echo, or carriage return echo mode, the associated data in the submailbox includes the delay timing table to be used (see Appendix A).

A delay timing table is shared among two or more channels using the same delay values. Delay timing tables are allocated and chained together; the first one is pointed to by .crdly in the system communications region. The format of the delay timing table is defined by the dlytbl macro. It includes forward and backward pointers, a reference count indicating how many channels are sharing it, and the following values:

dl.lf
: is the number of delays to be sent with a linefeed.

dl.cr
: is 512 times the number of delays to be sent with a carriage return for each column traversed. The formula for determining the total number of delays to be sent with a carriage return is:

    3 + (dl.cr*ncolumns)/512

dl.tba
: is the minimum number of delays to be sent with a horizontal tab.

dl.tbb
: is 512 times the number of additional delays to be sent with a horizontal tab for each column traversed. The formula for determining the total number of delays to be sent with a horizontal tab is:

    dl.tba + (dl.tbb*ncolumns)/512

dl.bs
: is the number of delays to be sent with a backspace.

dl.ff
: is the number of delays to be sent with a formfeed.

Other Blocks

Blocks allocated from the buffer pool used for purposes not discussed above are described in the sections on the modules that use them.

SECTION 11

SCHEDULER

The FNP scheduler is responsible for controlling the execution environment of the FNP. The scheduler consists of three main parts: the master dispatcher, the secondary dispatcher, and the timer management routines.

MASTER DISPATCHER

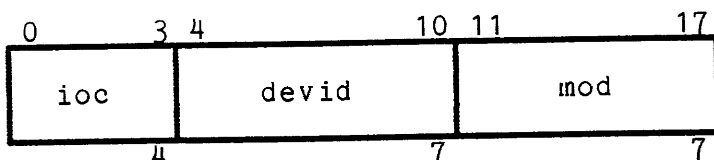
The master dispatcher is normally run in response to an interrupt. Interrupts in the FNP are handled by setting bits in interrupt cells at location 400-417(8). These cells are maintained by the hardware and scanned in a priority order. Whenever one of these bits is found to be on, the interrupt is presented to the processor, which computes an address based on which cell is set. This address is in the range 0-400(8), which represents the 256 possible interrupt cells in the FNP. If the processor is not inhibited, it executes a "tsy" indirect to the computed address. This location contains another level of indirect addressing, to the interrupt processing routine. However, if all interrupts of a certain class (e.g., HSLA or LSLA interrupts) go to a common routine, no information as to which device caused the interrupt is available. To solve this problem, the location in the interrupt vector contains not the address of the interrupt processing routine, but instead the address of a three-word jump table. This table has the following format:

zero
tsy interrupt_hndlr
coded word

There is a separate table for each interrupt vector address (0-400(8)), and these tables are set up by the hardware managers that run the device. Thus, when an interrupt occurs, the hardware will tsy indirect through the interrupt vector to the 3-word jump table, store the instruction counter (IC) at the time of the interrupt in the first word, and execute the tsy in the second word. This stores the address of the third word of the jump table in the first word of the interrupt handler, so the software can determine where the interrupt occurred and which device caused it.

In general, the tsy instruction in the second word of the jump table is a tsy to invp (the interrupt vector processor), an entry in the scheduler. This enables the scheduler to be invoked on each interrupt, and allows interrupt priorities to be assigned and enforced by the software.

The third word of the jump table, abbreviated 3wjt, has the following form:



where:

1. ioc
   is the IOM channel number of this device.

2. devid
   is a device specific identifier for multiple device channels (e.g., HSLA number and subchannel).

3. mod
   is a scheduler module number for the module to be called for this interrupt.

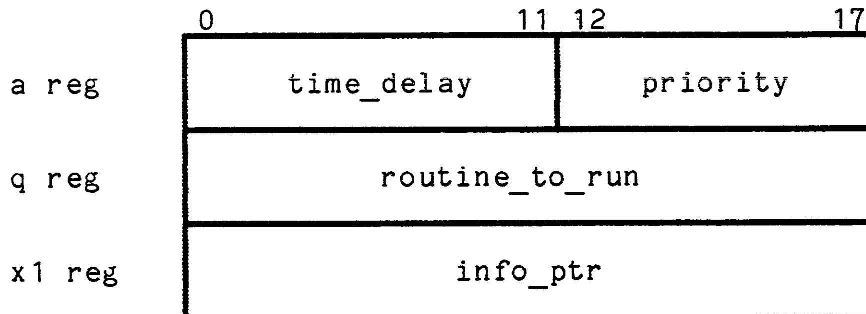
When invp is entered, it saves the third word of the jump table, the IC at the time of the interrupt, and all machine registers. It then enters the master dispatcher, which searches its tables looking for the highest priority interrupt. If the new interrupt is highest priority, it is run first; otherwise, the previous interrupt routine is restarted. Note that this scheme allows the actual interrupt service routine (e.g., hintr for HSLAs) to run uninhibited, to allow higher priority interrupts to occur. An interrupted occurrence of an interrupt handler is always allowed to complete before a new instance of the same interrupt handler is run.

When the interrupt handler is finished, it returns to the master dispatcher (via a tra mdisp) and the master dispatcher clears this entry from its tables and starts a search for a new routine to run. The last entry in the dispatcher's table is an entry for the secondary dispatcher. If any routines have been queued to be run, this entry is marked active. Thus, if no interrupt routines are to be run, the secondary dispatcher runs a queued routine.

This scheme causes the secondary dispatcher to be considered as if it were the lowest possible priority interrupt routine. If the secondary dispatcher, or a routine it is running, is interrupted, the master dispatcher saves all of the information as before, but when it is about to restart the queued routine, control is not returned to the point of interruption but instead to a secondary entry point in the secondary dispatcher. The secondary dispatcher can then reexamine its queues and run the highest priority routine and complete the interrupted routine at a later time.

SECONDARY DISPATCHER

Most interrupt routines called by the master dispatcher collect information about the interrupt and queue a routine to process it. Queueing of a routine is done via the scheduler entry dspqur. This entry gets its arguments in three registers as follows:



where:

1. time_delay
   is the delay, in seconds, before the routine is to be run.

2. priority
   is a scheduler priority (explained below).

3. routine_to_run
   is the address of the routine to be executed when its turn comes.

4. info_ptr
   is passed to the routine in index register 1 at the time it is run.

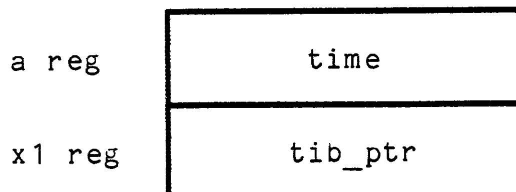
The time_delay may be zero, indicating that the routine is to be run as soon as possible. The priority is one of three groups: high (0-7), medium (10-17), or low (20-27). It is very important to note that the scheduler does not preempt a routine in a group with another routine in the same group, i.e., a routine started in the high group always runs to completion before any other routine is run. This implies that if a routine is scheduled in the medium priority group and an interrupt occurs that schedules a high priority routine, the high priority routine is run immediately. Thus, a medium priority routine cannot use any of the same storage or subroutines (unless the subroutine is inhibited) as a high priority routine. The current implementation of Multics Communication System uses only the high priority group, to avoid problems caused by two routines of different priority groups calling the same subroutine.

TIMER MANAGEMENT

Two uses are made of the timer management mechanism in the scheduler: the delayed queueing function, and terminal control timing functions.

The delayed queueing function is invoked via a call to dspqur, the dispatcher queueing routine, as described above. When dspqur notices that a delay has been specified, a queue entry is made in a special time delay dispatch queue, and the clock is updated if required.

Terminal control timing functions are handled from the control_tables by the interpreter via the scheduler entry setime. Two arguments are passed to setime as follows:



where:

1. time
   is the number of seconds to wait before a timeout occurs, and can be zero to reset the current timer for this channel.

2. tib_ptr
   is the address of the terminal information block (TIB) for the channel.

When setime is called, it searches its current queue for an entry for the requested TIB, and frees it from the list if one is found. It then makes a new entry in its special TIB queue for this TIB. The clock is updated if necessary.

The FNP clock is a 36-bit simulated real-time clock that counts the number of milliseconds since the bootload of the FNP. This clock is maintained by using the interval timer of the FNP to count real milliseconds. The simulated real-time clock always contains the time at which the interval timer will run out; i.e., when the interval timer runs out, the simulated real-time clock contains the current time. If no timers are required by either the TIB timer or the delay queue, then the inverval timer is set to its maximum value (2**18 milliseconds, approximately 4.4 minutes) and this amount is added to the simulated real-time clock.

When either of the timer routines is called, it gets the current time by subtracting the interval timer from the simulated real-time clock. The routine then adds the number of milliseconds of delay required by its caller and saves this information as the real time at which timeout is to occur. The delay queue manager keeps 6-word queue entries for each delayed routine as follows:

forward pointer
routine addr
real time clock value (two words)
register x1
priority

These entries are threaded with the earliest time first. The TIB timeout manager, setime, keeps 2-word queue entries, a forward pointer and the TIB address, with the clock value stored at t.time. The TIB timeout manager also threads the entries with the earliest time first.

Updating the clock consists of inspecting the first entry on the list just rethreaded and comparing the time in that entry to the simulated real-time clock. If it is earlier, the simulated real-time clock and the interval timer are updated to the earlier value.

When the interval timer runs out, both lists are searched for eligible timeouts. For each entry selected dspqur is called (with a zero delay value) and the entry is freed. Then the first entries of the two lists are compared to find the earliest time value and the simulated real-time clock and interval timer are updated with the new value. If no entries are present in either list, the time is set to the maximum.

ELAPSED TIME METERING

The elapsed timer is used to meter the relative amount of time the FNP is idle, and optionally to record the parts of the system in which the most execution time is spent. The elapsed timer is set by software to run out and thus generate an interrupt every 50 milliseconds, at which time the elapsed time runout handler is executed. The 50-millisecond interval can be changed by use of the sample_time request to the debug_fnp command.

Idle Time Metering

The elapsed time runout handler examines the instruction counter at the time of the interrupt to see if it contains the address of the "dis" instruction in the master dispatcher that is executed if there is nothing to do. If the dis instruction address is present, an "idle" counter is incremented by one; otherwise, a "busy" counter is incremented. The idle_time request to the debug_fnp command (see Appendix B) determines the amount of idle time by comparing these two counters.

Instruction Counter Sampling

If the ic_sampler module is included in the core image, the elapsed timer runout handler calls an entry in it, icmon. If instruction counter sampling has been enabled by the debug_fnp request line "ic_sample start", icmon increments a "bucket" associated with the 16-word range of addresses in which the instruction counter at the time of the interrupt falls. This information can be examined by means of the ic_sample request to the debug_fnp command, as explained in the description of the debug_fnp command in Appendix B.

SECTION 12

TERMINAL AND LINE CONTROL

Terminal and line control in Multics Communication System is managed by a set of control tables that consists of macros defining operations to be performed for each configured communications channel. The macros do not generate executable FNP instructions; rather, each macro generates a block of data (referred to as an operation block or "op block") which is recognized by a program called the interpreter. Loosely, a set of op blocks is executed on behalf of some channel; what this really means is that the interpreter is invoked to perform the operations specified by those op blocks.

ORGANIZATION OF THE CONTROL TABLES

Division Into Modules

The Multics Communication System control tables consist of several modules, and additional modules may be added as necessary. The main module, control_tables, contains the op block macros for the most commonly used line types, as well as tables defining various attributes of each line type. The other modules contain op blocks for various special-purpose line types or special types of line-control operations; these modules are optional and may be omitted from the Multics Communication System core image (as described in Section 17) if the relevant line type or special function is not used. The following supplementary control_tables modules are currently implemented:

acu_tables
: performs "dial out" operations to an automatic call unit (ACU)

autobaud_tables
: performs automatic baud-rate detection on asynchronous HSLA subchannels

ards_tables
: controls an ARDS-like terminal on a Bell 202C modem

t202_tables
: controls a TermiNet 1200 terminal on a Bell 202C6 modem

g115_tables
: implements the G115 synchronous line protocol (RCI)

vip_tables
: controls a VIP 7705 display terminal in nonpolled operation

bsc_tables
: controls a bisync binary synchronous subchannel of an HSLA

polled_vip_tables
: controls a VIP 7700 or VIP 7760 subsystem in polled operation

ibm3270_tables
: controls an IBM Model 3270 controller and associated displays and printers (bsc_tables must also be included)

Tables Included in the control_tables Module

HEADER

At the base of control_tables is an array of pointers to tables that other parts of Multics Communication System need to be able to find. This array consists of the following:

word 0
: address of the first op block to be executed for all channels

word 1
: address of an array of pointers to device info tables. This array is indexed by line type to find the device info table itself (described below)

word 2
: unused

word 3
: address of the device-type/speed table (described below)

word 4
: address of the first op block to be executed when reading a terminal's answerback

DEVICE INFO TABLE

There is a device info table for each line type. The device info table has the following format:

0 8	9 17
seq1	seq2
unused	
keyboard address	
printer address	
flags	
nl	cr
tab	bs
upshift	downshift
break list indicator	break1
break2	etc.

where:

seq1, seq2
: are the two sequence characters to be used in alternate messages to a synchronous channel; they are ignored for an asynchronous channel.

keyboard address
: is the address of the default keyboard addressing string for this line type.

printer address
: is the address of the default printer addressing string for this line type.

flags

indicate the default settings of certain flags in the terminal information block (TIB) and the software communications region:

bit 17

is the default setting of tfctrl (keyboard and printer addressing required)

bit 16

is the default setting of sffsct (default CCT is short--ignored for LSLA channels)

bit 15

is the default setting of tfsftr (terminal uses case-shift characters)

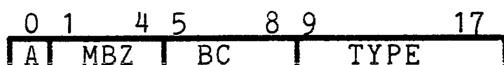
The following six characters are used in determining column position as a result of input or output. A character of all ones means that no applicable character exists for the specified line type.

nl	is the newline character
cr	is the carriage return character
tab	is the horizontal tab character
bs	is the backspace character
upshift	is the uppercase shift character
downshift	is the lowercase shift character

The remainder of the device info table is the break list, used for identifying break (end-of-message) characters on an LSLA channel; it is ignored for HSLA channels. The indicator is either 775 (octal), which indicates that break character status should be signalled in response to the input character immediately following the character specified in break1, or it is the number of characters (1 to 7) in the break list; break character status is signalled when one of these characters is input.

DEVICE-TYPE/SPEED TABLE

This table is used at initialization time to derive the default line type of each channel from the supplied baud rate code. Each entry is one 18-bit word in the following format:



where:

1. A
   is 0 for an asynchronous channel or 1 for a synchronous channel.

2. BC
   is the baud rate code, encoded as in an HSLA table entry (see Section 10).

3. TYPE
   is the default line type associated with a channel having the specified synchronous attribute and baud rate.

ADDRESSING STRINGS

The addressing strings pointed to by the device info table (see above) are strings of up to four characters used to "address" (i.e., enable) the keyboards and printers of certain terminals. Normally, such strings are required only for IBM 1050- and 2741-like terminals; the strings provided for the "ASCII" line type are generally only used for Teletype Model 37 terminals. The use of these strings is determined by the setting of the TIB flag tfctrl, which can be turned on or off at the request of the CS by means of an "Alter Parameters" mailbox operation (see Appendix A).

CONTROL TABLE INTERPRETER

The interpreter is the program that performs the operations described by the "executable" portion of the control tables, i.e., the op blocks. Whenever the interpreter is invoked on behalf of any channel, it uses the TIB whose address is in index register 1; t.cur in this TIB contains the address of the first op block to be executed, which must be a wait op block. (See Section 10 for a description of the TIB.) The interpreter passes over the specified op blocks, either proceeding from one to the next or "branching" elsewhere in the tables, depending on the op blocks being processed, until another wait or a waitm op block is encountered. At this point, t.cur is updated to point to the new block (wait) or left as it was on entry (waitm), and the

interpreter returns to its caller. The individual op blocks and their effects are described later in this section.

The interpreter may be entered at any of four different entry points, representing four different types of event: timeout, output, test-state, or status. These four event types are discussed in somewhat more detail below.

Timeout

A timeout occurs when a timer started by a setime op block runs out. A timer interrupt is processed by the scheduler (see Section 11), which causes the interpreter to be invoked at the entry point itime. Op block execution begins at the first branch address specified in the wait block pointed to by t.cur; if the address is zero, the interpreter returns without doing anything. If a timeout address is specified in a wait block, the block should be preceded by a setime op block, so that a timeout is in fact possible.

Output

When output for a channel arrives from the CS, and output is not currently being sent to that channel (as indicated by the TIB flag tfwrit), dia_man calls the interpreter entry point iwrite to start sending output to the channel. (See the discussion of dia_man in Section 13.) Op block execution begins at the second branch address specified in the wait block pointed to by t.cur; if the address is zero, the interpreter returns without doing anything.

Test-state

When the CS sends a submailbox containing one of a variety of WCD operation codes (see Appendix A) intended to change the state of a channel, dia_man turns appropriate TIB flags on or off and calls the interpreter at the entry point itest. Op block execution begins at the third branch address specified in the wait block pointed to by t.cur; if the address is zero, the interpreter returns without doing anything. The normal action by the control tables at a test-state branch is to test any TIB flags that are of interest in view of the current state of the channel.

Status

When hsla_man or lsla_man detects any of several kinds of change in the status of a channel, it calls the interpreter at the entry point istat. A word describing the latest status of the channel is passed in the A register. The kinds of status

change reported in this manner include changes in data-set status, the receipt of a break character (as defined by either the break list or the CCT), and various kinds of software status generated by contrl or cmd op blocks. The meanings of the individual bits in the status word are described later in this section.

When the interpreter is called at istat, it examines the op blocks immediately following the wait block pointed to by t.cur; in general, each wait block is followed by one or more status blocks. Each status block contains a word of "on" bits, a word of "off" bits, and a branch address. A status block is "satisfied" by the status word passed in the A register if all the 1-bits in the "on"-word of the status block are on in the status word and all the 1-bits in the "off"-word of the status block are off in the status word. (Zero bits in the status block are ignored.) The status blocks are examined in order to see if any of them are satisfied by the supplied status word; as soon as one is found that is satisfied, op block execution proceeds starting at the branch address specified in the satisfied status block. If the interpreter finds a nonstatus block before finding any satisfied status blocks, it returns without doing anything (the status is effectively ignored).

STATUS AND CONTROL BITS

The meanings of the bits used in status op blocks, status words passed to istat, and cmd and contrl op blocks are described below. These bits are defined by the csbits macro; the tconst macro defines the alternate forms of their names that are normally used in control tables.

Status Bits

Bits 0-3 each have two different meanings: one for a channel connected to an automatic call unit (ACU), the other for a binary synchronous (bisync) channel.

Bit position	Mask	Name	Meaning
0	400000	ads	ACU: raised data set status
		bscdmk	bisync: delayed marker
1	200000	acr	ACU: abandon call and retry
		bscmrk	bisync: marker status
2	100000	dlo	ACU: data line occupied

Bit position	Mask	Name	Meaning
		bscrcr	bisync: receive block termination
3	040000	pwi	ACU: power indicator
		rcvto	bisync: receive timeout
4	020000	xte	transfer timing error
5	010000	parity	parity error
6	004000	exh	"exhaust" status: indicates that absolute input buffer limit has been reached for the channel
		ring	ring indicator
8	001000	brkchr	break character received
9	000400	break	line break received (e.g., INTERRUPT button pressed)
10	000200	prexh	"pre-exhaust" status: indicates that initial input buffer limit has been reached for the channel
11	000100	term	terminate status (generated by control tables)
12	000040	marker	marker status (generated by control tables)
13	000020	st	status requested (generated by control tables)
14	000010	suprec	supervisory receive
15	000004	dsr	data set ready
16	000002	cts	clear to send
17	000001	cd	carrier detect

Bits 11-17 correspond to bits 11-17 of the TIB status word, t.stat.

Control Bits

These bits are specified in the contrl op block or in the cmd sub-operation of the dcwlst op block. (The contrl macro generates a dcwlst op block consisting of one cmd sub-operation. See the descriptions of the individual op blocks later in this section.) A cmd sub-operation usually contains one word of control bits, but may optionally contain a second word.

FIRST WORD

Bit position	Mask	Name	Meaning
0-2	100000		identifies sub-operation as CMD
3	040000		not used
4	020000	rrts	reset request-to-send
5	010000	srts	set request-to-send
6	004000	strm	send terminate status (reflected as "term" in subsequent status)
7	002000	smark	send marker status (reflected as "marker" in subsequent status)
	001000	sbrk	send line break to terminal
	000400	stat	send status to control tables (reflected as "st" in subsequent status)
10	000200	rsup	reset supervisory transmit
	100	ssup	set supervisory transmit
12	000040	rdtr	reset data terminal ready
13	000020	sdtr	set data terminal ready
14	000010	rxmit	reset transmit mode
15	000004	sxmit	set transmit mode
16	000002	rrec	reset receive mode
17	000001	srec	set receive mode

SECOND WORD

Bit position	Mask	Name	Meaning
0-2	500000		identifies second word of CMD sub-operation
3-15			not used
16	000002	rcrq	reset call request (ACU)
17	000001	scrq	set call request (ACU)

OP BLOCKS

The individual op blocks used in the control tables are described below. For each op block, the syntax of the corresponding macro is given, followed by a description of its effect and a schematic representation of the generated code.

TIB Extension Addressing

Each TIB may have a TIB extension associated with it. This extension is dynamically allocated and freed with the getext and retext op blocks. The extension may be used to hold either character variables, word variables, or both. Each variable type has its own addressing conventions. Character addresses are values in the range 460(8) to 477(8). These addresses correspond to the first 16 characters (8 words) of the TIB extension. There can be at most 16 TIB character variables, and all must be located in the first 8 words of the TIB extension. Word addresses are in the form of a negative word offset: the first word of the extension is -1, the second -2, etc. TIB character variables are used by op blocks such as setchr and inscan. TIB word variables are used by op blocks such as setlcl, tstlcl, etc.

Defining the format of the TIB extension can be simplified through the use of the tibex macro. This macro will equate a label to the next available word or character address and keep a tally of words used. The general format is:

```
tibex <label>,<type>
```

where <label> is the name of the address to be defined, and <type> is either "char" or "word".

For example:

```
     ex     a,char
  tibex     b,char
  tibex     c,char
  tibex     d,word
  tibex     e,char
```

will equate a to 460, b to 461, c to 462, d to -3, and e to 466. In addition, the variable tibxsz will be set to 4, which is the correct number of words to allocate with the getext macro.

Op Block Summary Lists

The op blocks described in this section are listed below, sorted first alphabetically and next by octal identifier. The page number on which each op block is described is given after the name and identifier of that op block.

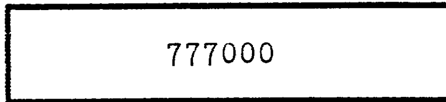
Name	Identifier	Page
acntr	777024	12-22
addlcl	777055	12-32
bkptop	777064	12-38
bldmsg	777032	12-27
calasm	777063	12-36
calsub	777036	12-24
ckinpt	777043	12-30
clrflg	777011	12-20
clrlcf	777060	12-33
cmd		12-15
cmpchr	777035	12-24
config	777042	12-28
contrl		12-18
dcwlst	777005	12-15
dmpmsg	777053	12-27
dmpout	777013	12-20
dmprpy	777046	12-30
dumpin	777033	12-27
echo	777051	12-31
getext	777026	12-22
gocase	777075	12-39
goto	777001	12-13
gotov	777074	12-38
gotype	777007	12-19
gtinpt	777044	12-30
holdot	777040	12-28
ifhsla	777041	12-28
iftype	777002	12-13
input		12-16
inscan	777030	12-26
linctl	777065	12-38
linsta	777066	12-37
meter	777015	12-25
nullop	777070	12-38
output		12-17
outscn	777031	12-26
prepnl	777047	12-31
punt	777000	12-13
rdtly		12-16
replay	777045	12-30
retext	777027	12-23
retsub	777037	12-24
scntr	777023	12-21
sendin	777017	12-25
setcct	777052	12-31
setchr	777034	12-23
setflg	777010	12-19
setime	777006	12-18
setlcf	777057	12-33
setlcl	777054	12-32
setlcv	777062	12-34
settmv	777072	12-35
setype	777022	12-21
signal	777014	12-22
status	777004	12-15
stpchn		12-18
tcntr	777025	12-22
tstflg	777012	12-20
tstglb	777021	12-25
tstlcf	777061	12-34
tstlcl	777056	12-33
tstlcv	777067	12-35
tstrpy	777050	12-31
tstwrt	777020	12-25
unwind	777071	12-38
wait	777003	12-14
waitm	777016	12-21

The following list sorts the op block by octal identifier (those having no identifier are included at the end of the list). Again, the page number on which the op block is described is given after the octal identifier and name of that op block.

777000	punt	12-13
777001	goto	12-13
777002	iftype	12-13
777003	wait	12-14
777004	status	12-15
777005	dcwlst	12-15
777006	setime	12-18
777007	gotype	12-19
777010	setflg	12-19
777011	clrflg	12-20
777012	tstflg	12-20
777013	dmpout	12-20
777014	signal	12-22
777015	meter	12-25
777016	waitm	12-21
777017	sendin	12-25
777020	tstwrt	12-25
777021	tstglb	12-25
777022	setype	12-21
777023	scntr	12-21
777024	acntr	12-22
777025	tcntr	12-22
777026	getext	12-22
777027	retext	12-23
777030	inscan	12-26
777031	outscn	12-26
777032	bldmsg	12-27
777033	dumpin	12-27
777034	setchr	12-23
777035	cmpchr	12-24
777036	calsub	12-24
777037	retsub	12-24
777040	holdot	12-28
777041	ifhsla	12-28
777042	config	12-28
777043	ckinpt	12-30
777044	gtinpt	12-30
777045	replay	12-30
777046	dmprpy	12-30
777047	prepnl	12-31
777050	tstrpy	12-31
777051	echo	12-31
777052	setcct	12-31
777053	dmpmsg	12-27
777054	setlcl	12-32
777055	addlcl	12-32
777056	tstlcl	12-33
777057	setlcf	12-33
777060	clrlcf	12-33
777061	tstlcf	12-34
777062	setlcv	12-34
777063	calasm	12-36
777064	bkptop	12-38
777065	linctl	12-37
777065	linsta	12-37
777067	tstlcr	12-35
777070	nullop	12-38
777071	unwind	12-38
777072	settmv	12-35
777074	gotov	12-38
777075	gocase	12-39
	cmd	12-15
	input	12-16
	rdtly	12-16
	output	12-17
	contrl	12-18
	stpchn	12-18

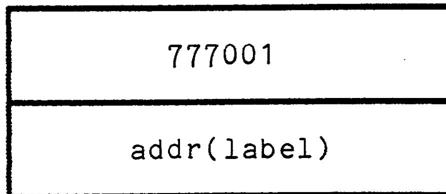
Op Block: punt

Causes the FNP to crash.



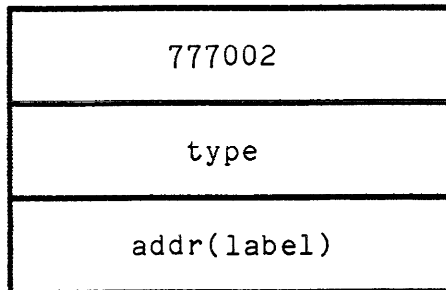
Op Block: goto <label>

Performs an unconditional branch to the op block specified by <label>.



Op Block: iftype <type>,<label>

Performs a "goto <label>" if the line type is equal to <type>.



Op Block: wait <timeout label>,<write label>,<test state label>

This op block causes the interpreter to wait for some event to happen on the channel before proceeding. There are 4 possibilities:

1. A "goto <timeout label>" is performed if the timer for the channel runs out. (see setime)

2. A "goto <write label>" is performed if the FNP receives output data from the CS for the channel

3. A "goto <test state label>" is performed if the FNP receives an order from the CS that changes certain TIB flags (tfhang, tflisn, tfercv, tfrabt, tfwabt, tfxhld, tfacu) or the global flags gbfhng or gbfup.

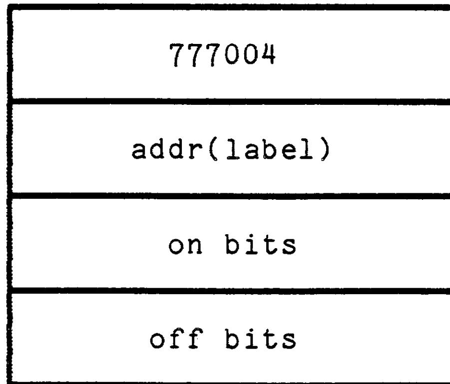
4. If status occurs for the channel, a check is made to see if status op blocks follow the wait op block. If so, they are checked to see what action is to be taken with the status.

Any or all of the operands may be 0. In this case, the event will be ignored if it happens.

777003
addr(timeout)
addr(write)
addr(test state)

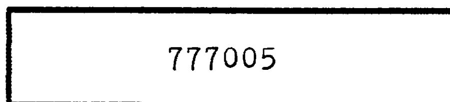
Op Block: status <on bits>,<off bits>,<label>

Status op blocks may only follow a wait op block. If status occurs for the channel, the current wait block is checked to see if status op blocks follow it. If so, each status op block is checked to see if it matches the status received. If so, a "goto <label>" is performed. The status is considered to match the op block if all the bits specified in the first operand are on and all the bits specified in the second operand are off. If no status blocks match the status received, the status is discarded and the channel reverts to the wait state. The meanings of the bits are described under "Status and Control Bits," above.



Op Block: dcwlst

The dcwlst op block is used to perform I/O on the channel. It must be followed by sub-op blocks that specify the I/O to be performed.



sub-op: cmd <commands>[,<secondary commands>]

This sub-op defines control operations to be performed on the channel. The second operand is only required if one of the secondary commands listed below is used. Commands valid in the first operand are:

srec	000001	set receive mode
rrec	000002	reset receive mode
sxmit	000004	set transmit mode
rxmit	000010	reset transmit mode
sdtr	000020	set data terminal ready
rdtr	000040	reset data terminal ready
ssup	000100	set supervisory transmit
rsup	000200	reset supervisory transmit

stat	000400	store status
sbrk	001000	send channel break
smark	002000	send marker status
sterm	004000	send terminate status
srts	010000	set request to send
rrts	020000	reset request to send

The following are valid commands in the second operand:

scrq	000001	set call request
rcrq	000002	reset call request

1	command flags
5	secondary flags

The second word is only generated when the <secondary commands> operand is used in the cmd sub-op.

sub-op: input <tally>,<char>

This sub-op causes characters to be read and discarded until either the tally runs out, or the character specified is encountered. It is valid only for LSLA channels.

2	tally	char

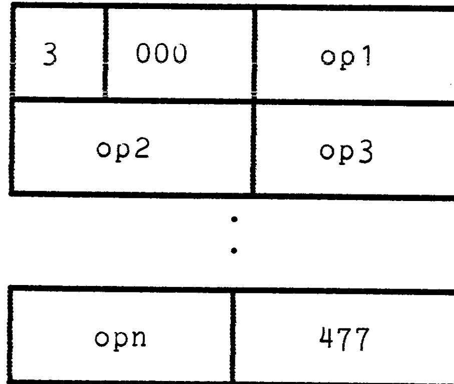
sub-op: rdtly <tally>

This sub-op block reads the number of characters specified by <tally>. It is valid only for LSLA channels.

4	tally	000

sub-op: output (<op1>,<op2>,...,<opn>)

The output sub-op inserts data into the output buffer chain to be written to the terminal. The <opi> consist of data or control codes.



All opi less than 400(8) are considered to be literal characters and are inserted directly into the output chain. The opi greater than 400(8) are control codes from the following list:

adprtr	401	Insert the terminal's printer addressing string.
adkybd	402	Insert the terminal's keyboard addressing string.
outmsg	403	Insert any data from the CS for this channel. If this control type is used, it must be the last operand in the output sub-op.
repeat	404	Inserts a single character a specified number of times. The operand following the repeat operand is the character to repeat; the next operand is the count.
	477	End of output sub-op. This code is inserted automatically when the output sub-op is generated. It may appear in either half of the word, depending on the number of operands that precede it.

Op Block: contrl <op1>[,<op2>]

This op block is used to perform a control operation on the channel. It is not a separate op block, but generates the following sequence:

```
        dcwlst
        cmd        <op1>[,<op2>]
```

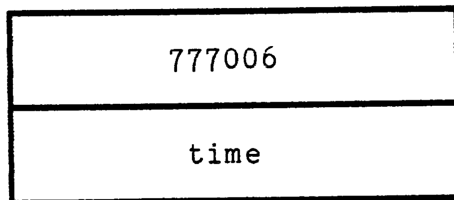
Op Block: stpchn

This op block is used to terminate all I/O on the channel. It is not a separate op block, but generates a calsub (see below) to the following sequence:

```
        contrl     rrec+rxmit+smark
        wait       0,0,0
        status     marker,0,.xxx.
.xxx.   retsub
```

Op Block: setime <time>

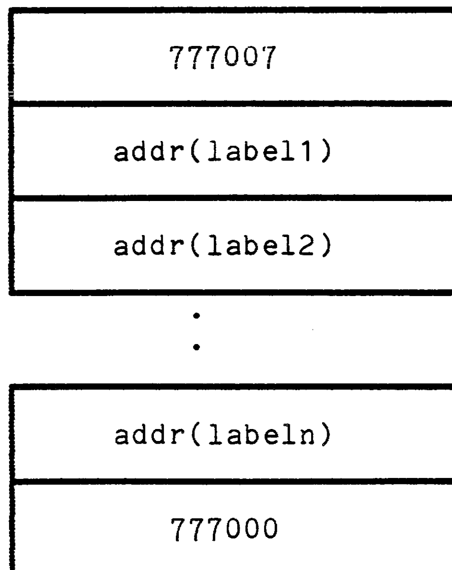
Sets a timer for the specified interval. If <time> is positive, the interval is in seconds; if negative, it is in milliseconds; if 0, any outstanding timer is disabled.



777006
time

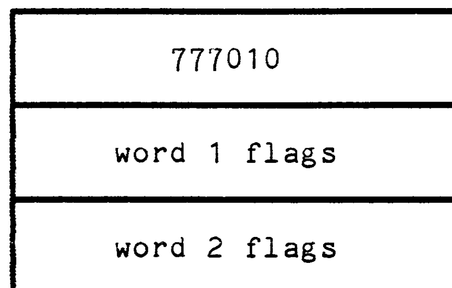
Op Block: gotype <label1>,<label2>,...,<labeln>

Performs a computed go to, using the line type as the index. If the line type is out of the range of labels specified, control falls through to the next op block.



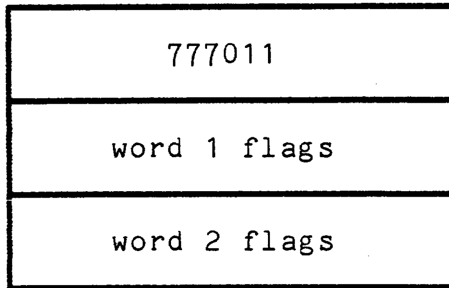
Op Block: setflg (<flag1>[,<flag2>,...])

Sets flags in the TIB. The flags specified may be defined either in t.flg or t.flg2.



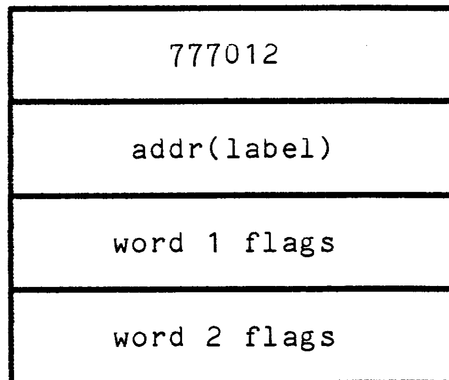
Op Block: clrflg (<flag1>[,<flag2>,...])

Clears flags in the TIB.



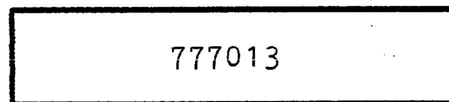
Op Block: tstflg (<flag1>[,<flag2>,...]),<label>

Performs a "goto <label>" if all the flags specified are on in the TIB.



Op Block: dmpout

Frees all the buffers in the output chain.



Op Block: signal <condition>

This op block allows the FNP to signal a condition to the CS. The following conditions are defined:

dialup	100	connection has been established
hangup	101	channel has been disconnected

sndout	105	ready for output
quit	113	line break condition
wrutim	114	"who-are-you" timeout
acupwi	120	no power to ACU
acudlo	121	data line occupied (ACU)
acuacr	122	dialout failed (ACU)
acung	123	invalid ACU request

777014
condition#

Op Block: waitm

The waitm op block returns the channel to a wait state at the most recently executed wait op block.

777016

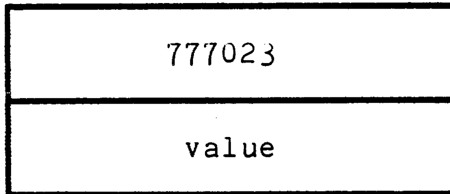
Op Block: setype <type>

The setype op block is used to set the line type in the TIB.

777022
type

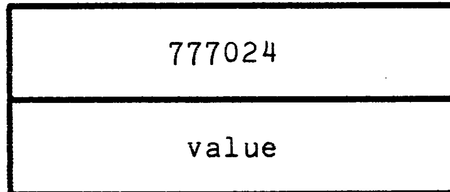
Op Block: scntr <value>

This op block is used to set the counter in the TIB.



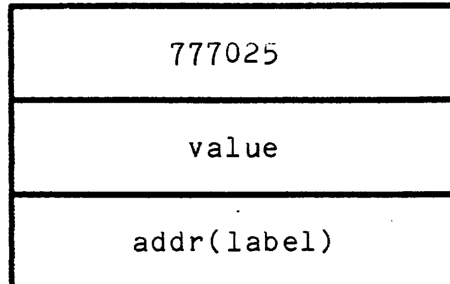
Op Block: acntr <value>

This op block is used to add a value to the counter in the TIB.



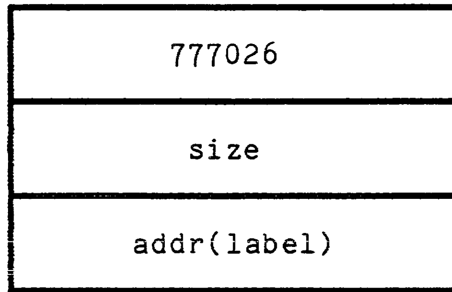
Op Block: tcntr <value>,<label>

This op block is used to test the counter in the TIB. If the counter is equal to <value>, a "goto <label>" is performed.



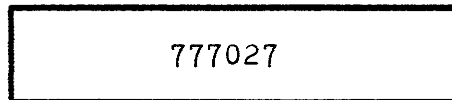
Op Block: getext [<size>],<label>

The getext op block allocates a TIB extension of <size> words in length. This may be used for temporary storage of characters, messages, etc. Only one TIB extension can be associated with each TIB, and it must be freed before a new extension can be allocated. If the extension cannot be allocated, a "goto <label>" is performed. If the <size> operand is omitted, the current value of tibxsz is used. This variable is defined automatically by the tibex macro (see "TIB Extension Addressing," above).



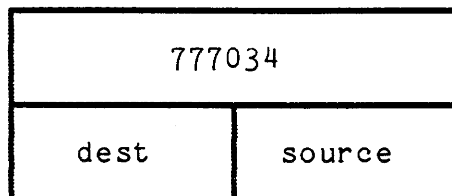
Op Block: retext

The retext op block frees the TIB extension allocated by a getext op block. A TIB extension must be present to use this op block.



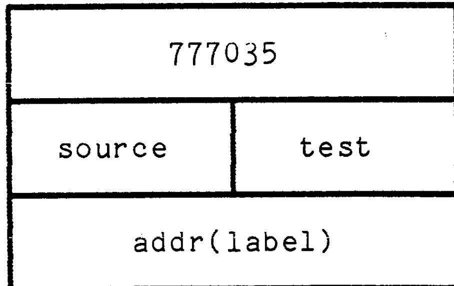
Op Block: setchr <destination>,<source>

The setchr op block stores a character in the TIB extension. The destination must be a TIB extension address in the range 460(8) to 477(8) (corresponding to a character offset in the TIB extension of 0(8) to 17(8)). The source may be another TIB extension address (if in the range 460(8) to 477(8)), or a literal character (if any other value). The TIB must have an extension allocated before this op block is used.



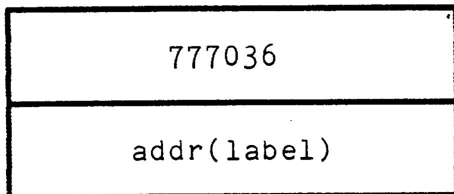
Op Block: cmpchr <source>,<test>,<label>

The cmpchr op block compares two characters and performs a "goto <label>" if they are equal. The source and test values may be either literal characters or TIB extension addresses. (See the setchr op block.)



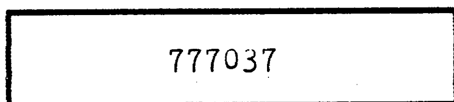
Op Block: calsub <label>

The calsub op block is used to call a subroutine within the control tables. The interpreter saves the address of the op block following the calsub and performs a "goto <label>". Only two levels of subroutine are allowed; a wait op block may not appear in an inmost-level subroutine (i.e., one which is called by another subroutine); a stpchn op block may not appear within any subroutine.



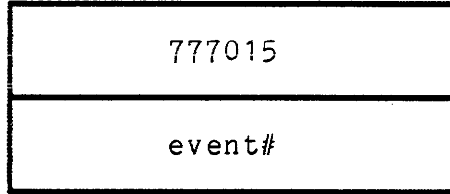
Op Block: retsub

The retsub op block is used to return from a subroutine called by a calsub op block. The interpreter gets the next op block from the word following the calsub op block.



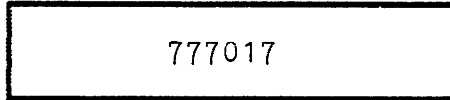
Op Block: meter <event>

This op block increments the meter assigned to <event>. It may be used to keep track of unusual occurrences. See the discussion of metering in Section 14.



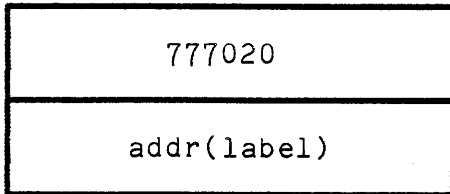
Op Block: sendin

This op block causes any pending input from the channel to be sent to the CS.



Op Block: tstwrt <label>

This op block causes a "goto <label>" if there is any pending output for the channel.



Op Block: tstglb <switches>,<label>

This op block performs a "goto <label>" if all the bits specified in <switches> are on in the global switch word used to maintain information of global interest to Multics Communication System. The following switches are defined:

gbfup	000001	CS is running
gbfhng	000002	all lines are to be disconnected as a result of a "blast hangup" command

gbfbla	000004	output is being sent to all channels for a "blast" command

777021
switches
addr(label)

Op Block: inscan <control>,<label>

This op block causes the current input chain to be scanned under control of the control string defined at the label <control>. If the scan fails, a "goto <label>" is performed. Control string formats are described later in this section.

777030
addr(control)
addr(label)

Op Block: outscn <control>,<label>

This op block is similar to the inscan op block except that it scans the current output chain.

777031
addr(control)
addr(label)

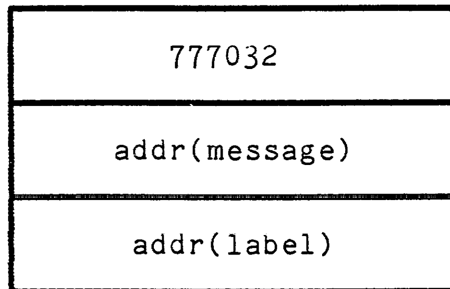
Op Block: bldmsg <message>,<label>

This op block causes the output message specified by the character string at the label <message> to be chained on to the head of the channel's current output chain. It is suitable for generating message blocks for synchronous line protocols. If the message cannot be built, a "goto <label>" is performed. Each character in the message is one of the following:

514 — turn on the "last buffer" flag in the last buffer of the message

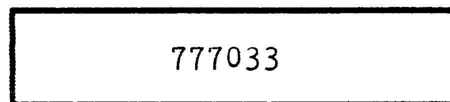
TIB extension character address — insert the character from the specified TIB extension address in the message

other — literal character to be inserted into the message



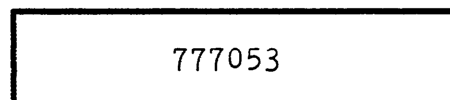
Op Block: dumpin

This op block causes any pending input for the channel to be discarded.



Op Block: dmpmsg

This op block causes the current input message (i.e., all input data up to and including a buffer with the "last" flag on) to be discarded.



Op Block: holdot

This op block causes the "hold buffer" flag to be set in every buffer of the current output message (i.e., all output buffers up to the first one with its "last buffer" flag set). It is used to prevent output buffers from being discarded when they have been transmitted, in case it is necessary to retransmit a message as a result of an error.

777040

Op Block: ifhsla <label>

This op block executes a "goto <label>" if the channel is configured on an HSLA.

777041
addr(label)

Op Block: config

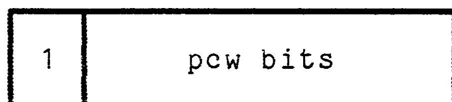
This op block causes an HSLA subchannel to be reconfigured. It must be followed by sub-op blocks specifying the details of the reconfiguration.

777042

sub-op: smode <pcw bits>

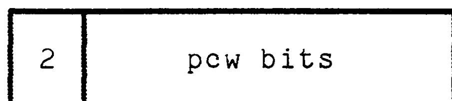
This sub-op causes the bits that are on in <pcw bits> to be turned on in the HSLA configuration PCW. The bits that may be specified in <pcw bits> are:

fg.icw	000001	two send ICWs
fg.lpr	000002	lateral parity receive
fg.lps	000004	lateral parity send
fg.lpo	000010	odd lateral parity
fg.5bt	000020	5-bit character mode
fg.6bt	000040	6-bit character mode
fg.7bt	000100	7-bit character mode
fg.8bt	000200	8-bit character mode



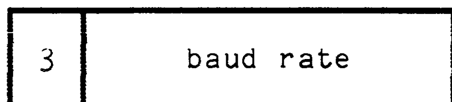
sub-op: rmode <pcw bits>

The rmode sub-op causes the bits that are on in <pcw bits> to be turned off in HSLA configuration PCW, where <pcw bits> are as described above for the smode sub-op.



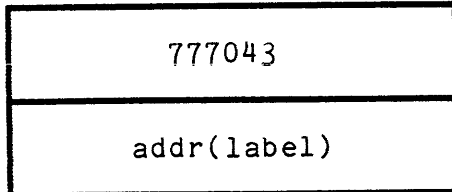
sub-op: baud <baud rate>

The baud sub-op specifies that the baud rate for the channel is to be set to <baud rate>. If <baud rate> is zero, the value in t.cntr (set by a scntr op block) is to be used.



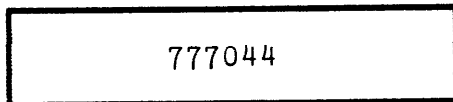
Op Block: ckinpt <label>

The ckinpt op block is used to see if the input chain for a channel ends in a partial line (i.e., the terminal is not at the left margin). It is useful in determining whether to wait before sending output to a channel in "polite" mode. If there is no partial line of input, a "goto <label>" is done.



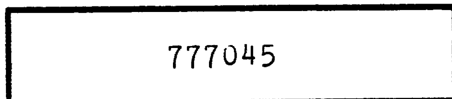
Op Block: gtinpt

This op block copies the current input chain into the "replay" chain pointed to by t.rcp. It is used for replaying interrupted input to a channel in "replay" mode.



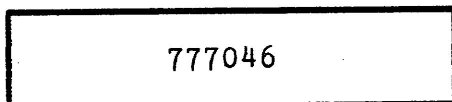
Op Block: replay

This op block sends the contents of the "replay" chain as output to the channel.



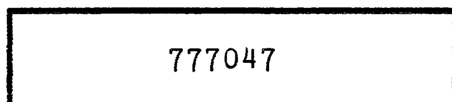
Op Block: dmprpy

This op block causes the contents of the "replay" chain, if any, to be discarded.



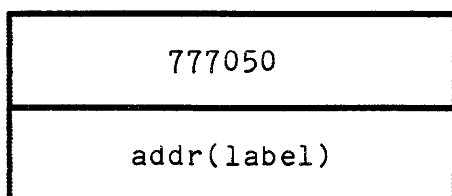
Op Block: prepnl

This op block is used to output a newline character before sending output that interrupts a partial line of input.



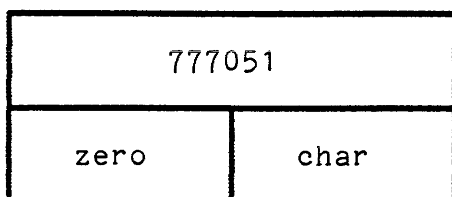
Op Block: tstrpy <label>

This op block performs a "goto <label>" if there is no replay chain pointed to by t.rcp.



Op Block: echo <char>

This op block causes the character specified by <char> to be inserted in the echo buffer for echoing at the next opportunity.



Op Block: setcct <cct_spec>

This op block causes an HSLA channel to start using the specified character control table (CCT). <cct_spec> may be the address of a CCT, or any of the following codes (defined in the cctdef macro):

scc.dl	0	delete the CCT currently in use
scc.df	1	use the default CCT for the current line type and modes
scc.bs	2	return to the base CCT when the table switch feature has been used (see discussion of CCTs later in this section)

777052
cct_spec

Op Block: setlcl <var>,<value>

This op block sets the local variable <var> to the value given. The variable may be either a TIB extension offset or an absolute FNP address.

777054
addr(var)
value

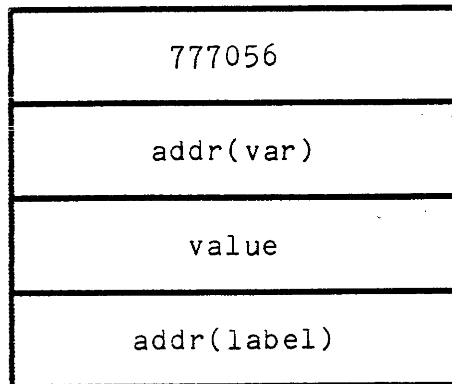
Op Block: addlcl <var>,<value>

This op block adds the <value> specified to the local variable <var>. The variable may be either a TIB extension offset or an absolute FNP address.

777055
addr(var)
value

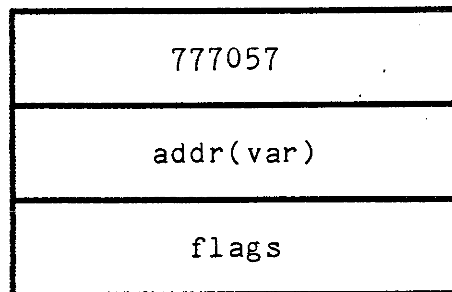
Op Block: tstlcl <var>,<value>,<label>

This op block compares the variable <var> with <value>, and if they are equal, does a goto <label>.



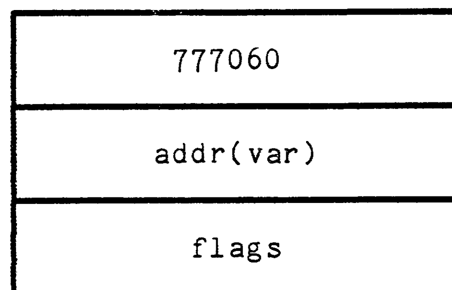
Op Block: setlcf <var>,<flags>

This op block turns on the bits specified by <flags> in the local variable <var>.



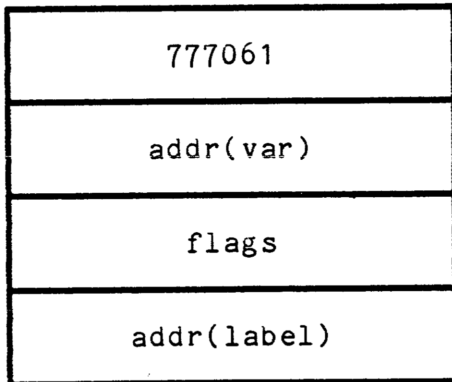
Op Block: clrlcf <var>,<flags>

This op block clears the bits specified by <flags> in the local variable <var>.



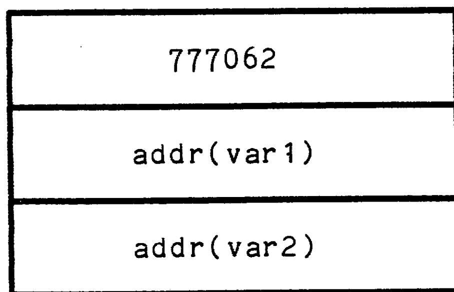
Op Block: tstlcf <var>,<flags>,<label>

This op block tests the bits specified by <flags> in the local variable <var>, and if they all are on, does a goto <label>.



Op Block: setlcv <var1>,<var2>

This op block sets the local variable <var1> to the value of local variable <var2>.



Op Block: tstlcv <var1>,<var2>,<label>

This op block compares the value of the local variable <var1> with that of the local variable <var2>; if they are equal, it performs a goto <label>.

777067
addr(var1)
addr(var2)
addr(label)

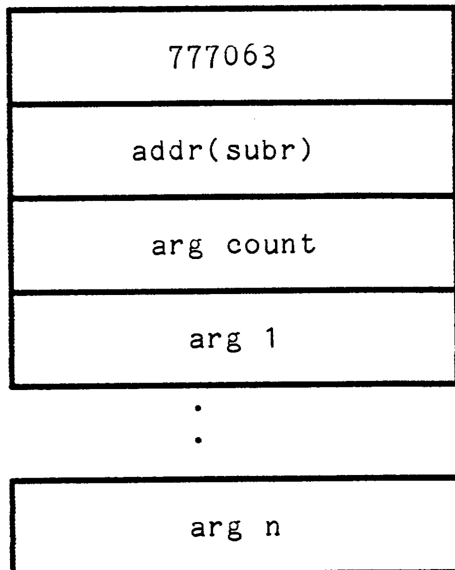
Op Block: settmv <var>

This op block sets a timer using the value of the local variable <var> for the time interval.

777072
addr(var)

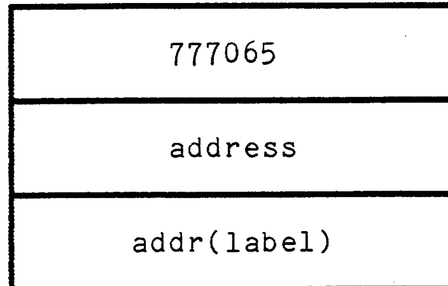
Op Block: calasm <address>,(<param1, ..., paramn>)

This op block calls a subroutine written in 355MAP assembler language. The subroutine is specified by its address in the first operand. If the subroutine requires arguments, they may be specified as a list in the second argument. The assembler subroutine is entered with the TIB address in index register 1, the argument count in index register 2, and the address of the argument list in index register 3. When the subroutine returns, it must either zero index register 2, which implies an inline return to the next op block, or load index register 2 with the address of the op block to return to.



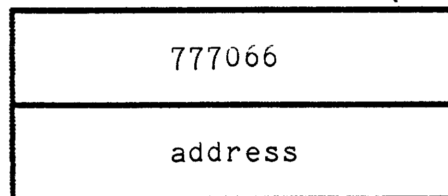
Op Block: linctl <address>,<label>

This op block checks to see if the current test-state event is the result of a "line_control" order from the CS, and if so stores the four words of data associated with the order at <address>, which can be either a TIB extension address (of word type) or an absolute FNP address. If no "line_control" order was sent, it performs a goto <label>.



Op Block: linsta <address>

This op block reports line status to the CS. The four words at <address> are sent to the CS and stored as line status for the channel; no I/O can be done on the channel until a "line_status" order is performed by the I/O module to pick up the status. <address> can be either a TIB extension address or an absolute FNP address.



Op Block: bkptop

This op block represents a breakpoint in the control tables. It is inserted by breakpoint_man when a breakpoint is set using debug_fnp. See the discussion of breakpoints in the description of the debug_fnp command in Appendix B.

777064

Op Block: nullop

This op block is a no-operation. Control proceeds to the following op block.

777070

Op Block: unwind

This op block clears any pending subroutine return addresses. It can be used in preparation for a "non-local" goto from within a subroutine to the outermost level of op block execution.

777071

Op Block: gotov <var>

This op block performs a goto to the label whose address has been stored in the local variable <var>.

777074
addr(var)

Op Block: gocase <var>,<value_label>,<goto_label>

This op block selects a branch address depending on the value of the local variable <var>. <value_label> identifies a list of character values generated by means of a chstr macro (described below under "Description of Scan Control Strings"); <goto_label> identifies a list of possible branch addresses. The value list is scanned from left to right until a value is found that is equal to the value of <var>; then a goto is done to the corresponding branch address. If no match for <var> is found, control proceeds to the following op block.

The list of addresses may be generated by means of the adrlst macro, as follows:

adrlst (label1, label2, ..., label*n*)

The values in the value list may be either character literals or TIB extension addresses.

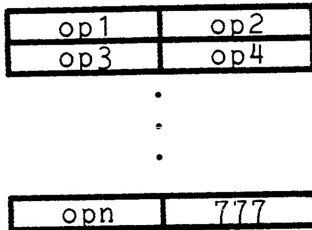
777075
addr(var)
addr(value_label)
addr(goto_label)

Description of Scan Control Strings

The scan op blocks work with a scan control string and the characters in either the input or output buffer chain. The control string consists of a series of characters that are interpreted to find what actions to perform. Each character of the form 5xx(8) is a code indicating an action to be performed. The rest of this section describes these actions in detail. Some of the action codes require one or more data characters as well. For these codes, the data characters follow the action code in the control string. A data character may be either a literal value or a TIB extension address. A TIB extension address is indicated by a data character in the range 460(8) to 477(8); when one of these codes is encountered, the data character is fetched from the TIB extension (character 0(8) through 17(8) respectively).

The control string for the scan op block is generated by the chstr macro, as follows:

chstr (op1,op2,...opn)



The end of string marker (777) is inserted automatically by the chstr macro after the last operand given. It may appear in either half of the last word, depending on how many operands precede it.

During the scan, a pointer is kept to the current character in the buffer chain. This pointer is undefined whenever the control tables are entered (i.e., when resuming after a wait block). The pointer must be initialized to the first character in the chain with the "rescan" control code. Most control codes either deal with the character designated by the current chain pointer, or move the pointer in some fashion.

The following codes may appear in the control string:

match 501
The next character in the control string is the match character. It may be either a literal value or a TIB extension address. If the next character in the chain is equal to the match character, the scan continues; otherwise, the scan terminates and the failure exit is taken.

search 502
The next character in the control string is the match character. The current chain is scanned looking for the match character. If it is not found, the scan fails. If it is found, the scan continues, and the chain pointer is left at the character that was found.

ignore 503
Bumps the chain pointer to the next character. The character skipped over is omitted from any block check calculation. If the pointer is already past the last character, an inscan will fail. An outscn,

however, increases the length of the output chain by 1 character and moves the pointer over the character.

sbcc 504
Sets a flag to start the block check calculation. All characters passed over in the chain (by ignore, search, etc.) are exclusive or'ed into the block check accumulator.

endchn 505
Sets the chain pointer to the last character in the last buffer in the buffer chain.

cbcc 506
Stops the block check calculation and compares the accumulated result with the next character in the buffer chain. If it matches, the scan continues. Otherwise, the scan fails.

cmask 507
This control requires two more characters in the control string. The first is a match character and the second is a mask. The match character and the next character in the buffer chain are both and'ed with the mask. If the results are equal, the scan continues; otherwise, it fails.

rescan 510
Resets the current buffer chain pointer to the start of the appropriate chain. This is normally the first operation in a scan control string.

strlrc 511
Starts the "longitudinal redundancy check" computation. It is identical in meaning to the "sbcc" control (above).

outlrc 512
(Used for output scans only) inserts the accumulated block check character at the character position indicated by the chain pointer.

cmplrc 513
Identical to cbcc.

seteom 514
Turns on the "last buffer in message" flag in the buffer into which the chain pointer points.

replac 515
Must be followed by a data character, which is either a literal character or a TIB extension address. The resulting character (the literal or the character at the TIB extension address) replaces the character addressed by the chain pointer.

cmplst 516
Must be followed by one or more data characters as specified above. The character addressed by the chain pointer is compared with each data character in turn; if it matches one of them, the remaining data characters are skipped and the scan continues. If the end of the control string or another scan control is encountered before the current character is matched, the scan fails.

movchr 517
Must be followed by a TIB extension address. The character addressed by the chain pointer is copied to the specified TIB extension address.

movmsk 520
Must be followed by two data characters, of which the first is a TIB extension address and the second is a mask. The character addressed by the chain pointer is and'ed with the mask and stored at the specified TIB extension address.

count 521
Indicates that a count of characters in the chain is to be accumulated at the TIB extension address specified by the next character in the control string. For every character passed over by a search, serch2, nxtchr, or ignore control, the value at the TIB extension address is incremented by 1.

serch2 522
Is the same as search (above) except that the search is made for either of the two data characters following the control. If either character is found in the chain, the chain pointer is set to address the character found; if neither character is found, the scan fails.

setbit 523
Must be followed by a character that is used as a mask. The mask is or'ed into the character addressed by the chain pointer.

offbit 524
Must be followed by a character that is used as a mask. For every 1-bit in the mask, the corresponding bit in the character addressed by the chain pointer is turned off.

chktrm 525

Must be followed by a data character, which is compared to the next-to-last character in the chain; the chain pointer is set to address this character. If the characters are equal, the scan continues; otherwise it fails. If there are fewer than two characters in the chain, the scan fails.

mvlst2 526

Must be followed by two TIB extension addresses. The next-to-last character in the chain is copied into the first TIB extension address, and the last character in the chain is copied into the second TIB extension address. The chain pointer is set to address the last character in the chain. If there are fewer than two characters in the chain, the scan fails.

nxtchr 527

Bumps the chain pointer to the next character in the same manner as "ignore", except that the character passed over is included in the block check calculation if an "sbcc" or "strlrc" control is in effect.

PROVIDING ADDITIONAL CONTROL TABLES

A site may provide additional special-purpose control tables in order to run some communications protocol not supported by the standard Multics Communication System. The appropriate way to do this is as follows:

1. Write a module using the op block macros described above;

2. Assemble it by means of the map355 command;

3. Add the name of the new module to the "Order" statement in the bindfile used by the bind_fnp command;

4. Create a new core image using the bind_fnp command;

5. Specify the pathname of the new core image in the "image" statement of the channel master file (CMF), and use the cv_cmf command to generate a new channel definition table (CDT).

The rest of this section contains information needed to write a useful control tables module (step 1 above). For more details on steps 2-4, see the descriptions of the map355 and bind_fnp commands in Section 17; for more details on step 5, see the <u>MAM Communications</u>, Order No. CC75.

Pseudo-ops and Data-defining Macros

The source of a control tables module should begin with a few 355MAP pseudo-operations and macros that define symbols required for proper operation of the op blocks that make up the main body of the control tables. Some of the pseudo-ops mentioned here serve simply to increase the readability of the listing; others, particularly "symdef" and "symref" pseudo-ops, are required for correct operation. All 355MAP pseudo-operations are described in the manual, DATANET355 Macro Assembler Program, Order No. BB98.

LISTING CONTROLS

The following sequence of pseudo-ops and macros is present at the beginning of all control tables modules supplied as part of Multics Communication System. It is advisable to use a similar sequence in installation-supplied modules.

```
                lbl       ,<full_name>
                ttl       <suitable page heading>
                editp     on
                pmc       off
                detail    off
                pcc       off
                pmc       save,on
        <name>  null
                start     <name>
```

where <full_name> is the name of the source segment (without the ".map355" suffix), <suitable page heading> is a heading that appears on every page of the listing, and <name> is the name of the module as it appears in dumps of the FNP (see Section 16); <name> is a 355MAP symbol, and accordingly cannot exceed six characters in length.

EXTERNAL SYMBOL DEFINITIONS

Any symbol defined in the module that is referenced in any other module must appear as the operand of a "symdef" pseudo-operation in the module in which it is defined. If the installation-supplied module replaces a standard module, any "symdef" pseudo-operations appearing in the standard module must also appear in its replacement. In any case, the label at which the tables are initially entered for each channel must appear in a "symdef" pseudo-operation. See also the discussion of "Interaction with the Main Control Tables Module" later in this section.

Any symbol referenced in the module that refers to a symbol defined in another module must appear as an operand of a "symref" pseudo-operation in the referencing module. (Such symbols must, of course, also appear in "symdef" pseudo-operations in the defining module.) The symbols most likely to be referenced by a control tables module are "begin," "hungup," "stpchn," and "error," all of which are defined in the main control tables module, control_tables. See the discussion of "Interaction with the Main Control Tables Module" later in this section for details.

INTERNAL SYMBOL DEFINITIONS

The following macros are required to define data that is referenced symbolically by the op block macros:

tib
: defines the fields and flags in the TIB.

csbits
: defines the bits used in contrl, cmd, and status op blocks.

tconst
: defines many useful constants, including frequently-used ASCII and EBCDIC characters.

CHARACTER CONTROL TABLES

If character control tables (CCTs) other than the standard ones provided in the control_tables module are needed, they must be provided in the installation-supplied module. To facilitate the coding of CCTs, the cctdef macro should appear; this macro assigns mnemonic names to the bit patterns for the most commonly-used CCT entries. Each CCT must be forced to a 0 mod 64 address by preceding it with a "base 64" pseudo-operation.

The setcct op block must be used in the control tables module to establish the use of this CCT. The implementation of some protocols may require the use of the table-switching feature, whereby the coding of a particular input character in the CCT may automatically cause the HSLA hardware to switch to a different CCT; in such a case, an array of from 2 to 4 contiguous CCTs are provided. Switching back to the original (base) CCT is accomplished either through the appearance of a character appropriately encoded in the new CCT, or by software by means of the setcct op block with a cct_spec of cct.bs (see the description of the setcct op block). The bsc_tables module contains examples of the use of this feature.

SUPPRESSION OF OP BLOCK EXPANSION

The remainder of the module consists of op blocks suitable for "execution" by the interpreter, along with "chstr" macros (described earlier in this section) for any scan or bldmsg control strings required. If a "pmc restore" pseudo-operation appears before the beginning of the "executable" op blocks, the 355MAP expansions of the macros do not appear in the listing; the absence of these expansions greatly increases the readability of the listing.

Interaction with the Main Control Tables Module

The addition of a new communications protocol may require minor changes to the main control tables module (control_tables); in any case, it is important to understand how the new module interacts with control_tables.

LINE TYPES

The control tables used to drive any channel are selected on the basis of its line type. In general, the correct way to add a communications protocol is to use one of the line types reserved for site-defined protocols: ASYNC1, ASYNC2, or ASYNC3 for an asynchronous protocol, or SYNC1, SYNC2, or SYNC3 for a synchronous protocol. The name of this line type can then be specified in the channel master file (CMF) for those channels using the new protocol. The device table for that line type is specified in control_tables, as explained earlier in this section.

Finally, entry into the new module is effected by means of a gotype op block at the label "start" in the control_tables module. The label on the starting op block of the added module must be the same as the symbol to which the gotype op block branches for the line type in question. This symbol must appear in a "symdef" pseudo-operation in the added module.

The symbols associated with the reserved line types are as follows:

line type	symbol
ASYNC1	a1star
ASYNC2	a2star
ASYNC3	a3star
SYNC1	s1star
SYNC2	s2star
SYNC3	s3star

ANSWERBACK READING

If the line type being replaced has an answerback (or who-are-you) capability, and the gotype op block at the label anslp in control_tables branches somewhere for the type in question, a corresponding label must be supplied. If a who-are-you sequence is to be sent to the channel, the op blocks to send it and receive the answerback must begin at this label (which is invoked automatically in response to a who-are-you request from the central system); if the who-are-you request is to be ignored, the following sequence should be used:

```
signal   wrutim
goto     <main wait block>
```

where <main wait block> is the label on the normal "idle" wait block described under "Programming Considerations" below. In general, synchronous line protocols do not include an answerback capability; since additional special-purpose line protocols are usually synchronous, the answerback function is not expected to be a concern to most sites.

USEFUL LABELS IN THE MAIN MODULE

Op block execution for each channel always starts at the label begin in control_tables. When a channel is disconnected, the module may return to begin in order to wait for a listen request from the CS; it is more usual, however, to go to hungup (also in the control_tables module) to turn off data terminal ready and wait for the listen request. Another label in control_tables that may be of interest is error; branching to this op block causes the FNP to crash, which might be the only recourse in case of unaccountable results indicating software errors. This can be a useful tool in debugging a new control tables module. The stpchn label is the address of the subroutine invoked by the stpchn op block.

Programming Considerations

At entry to the added module, a test should be made to ensure that the TIB flag tflisn is on, indicating that the CS is prepared to communicate with the channel. If the flag is off, the control tables should wait for a test-state event, which should return to the label begin in the main module. If the tflisn flag is on, the tables should proceed to initiate communication with the channel (usually by raising the DTR dataset lead and waiting for some combination of DSR, CD, and CTS).

Once the connection has been established, the action of the control tables will be driven by various events:

1. Requests from the CS, indicated by "test-state" events and the setting of various TIB flags;

2. Output from the CS, indicated by "write" events and the presence of an output chain;

3. Input from the channel, indicated by "status" events;

4. Abnormal conditions on the channel, also indicated by "status" events.

In most cases, the module contains an "idle" wait block, to be branched to when any particular transaction completes, where it waits for the first of these events that occurs. If some activity on the channel is required at specified time intervals, this wait block should be provided with a timeout branch as well.

REQUESTS FROM THE CS

The most usual requests from the CS are listen, hangup, dump input, and dump output. These are all effected by means of a test-state event, with the TIB flags tflisn, tfhang, tfrabt, or tfwabt, respectively, turned on. The control tables should take appropriate action in each case. The listen request is not generally sent to an active channel, and can usually be ignored; after a hangup request, the channel should be disconnected; dump input and dump output can be handled by dumpin and dmpout op blocks respectively. In case the test-state event occurs at a wait block that is not checking for it, these flags should be tested at appropriate times.

OUTPUT FROM THE CS

When output arrives from the CS, and output is not already in progress for the channel, a write event is signalled to the control tables; appropriate dcwlst op blocks should be executed to send the output to the channel, generally storing terminate status on completion. The wait blocks used in actually completing the I/O should not branch on write events; but before returning to the idle wait block, a tstwrt op block should be used to check for the presence of additional output.

INPUT FROM THE CHANNEL

A completed input message from the channel is indicated by brkchr status. The normal response to this status is to check the input message for validity (if necessary) and execute a sendin op block to ship the input data to the CS. Of course, if

the expected input is some kind of control message (e.g., acknowledgement of output), it can be processed in the control tables and discarded, rather than being sent to the CS.

ABNORMAL CONDITIONS ON THE CHANNEL

A wait block executed when the channel is in receive mode should always be followed by status blocks to check for various conditions. Loss of carrier may or may not be interesting, depending on the line protocol; loss of data set ready (DSR) generally indicates that the connection has been broken. "Pre-exhaust" status (prexh) is generated when 10 input buffers have been filled and no break character has been encountered; it is usually advisable to respond by shipping the accumulated input and continuing to accept further input.

Exhaust status (exh) is generated when an excessive amount of input has accumulated; in this case, the normal response is to temporarily shut off the channel (by means of a stpchn op block), and not re-enter receive mode until after output has been sent to the channel or a line break has been received. A line break is indicated by break status; it usually indicates that a user has generated an interrupt from a terminal, which should be echoed back to the CS by means of a signal quit op block. Input transfer timing error (xte) status may be generated if an input buffer could not be set up in time; it should be treated like exhaust status. Finally, channel hardware errors may be reflected as parity status; this may be either ignored or treated like·exhaust status.

LINE CONTROL AND LINE STATUS

The line control and line status mechanisms can be used to facilitate communication between a control tables module and either a user-ring I/O module or a ring-0 multiplexer module without the rest of Multics Communication System having to know anything about the specific protocol. The linsta (line status) op block passes 72 bits of arbitrary information to the CS, resulting in a "line_status" interrupt; this interrupt may be intercepted by a multiplexer module, but if it reaches tty_interrupt, the latter stores the status and sets a flag in the WTCB and wakes up the process using the channel (see Sections 3 and 4 for further information about interrupts). The information can be obtained by an I/O module by means of the line_status control operation (described below).

Similarly, if an I/O module or a multiplexer issues a "line_control" control operation (described below), 72 bits of associated data are sent to the FNP; the interpreter is called with a "test-state" event. The control tables can then pick up

the associated data by means of a linctl op block, provided the op block is executed in response to the same test-state event.

USER-RING I/O MODULES

A control tables module that implements a protocol not otherwise supported by Multics probably requires the cooperation of a special-purpose I/O module. (Examples of such combinations in system supplied software include the RCI protocol, implemented by g115_tables and the g115_ I/O module; and the binary synchronous protocol, implemented by bsc_tables and the bisync_ I/O module.) The type of cooperation involved may include a common understanding of message formats, so that the I/O module can prepare output messages in a format that the control tables expect, and conversely for input messages. It may also include the use of the line status and line control mechanism.

Once line status has reached the WTCB level, any attempt by the user process to do I/O on the channel (through calls to tty_read and tty_write) result in a status code of error_table_$line_status_pending being returned. This condition can only be cleared by issuing a "line_status" control operation; the info_pointer passed with the control call must point to a bit(72) variable which is to be filled in with the line status sent from the FNP. If a "line_status" control operation is issued when no line status is in fact pending, a status code of error_table_$no_line_status is returned.

To send arbitrary control information to the control tables, the I/O module issues a "line_control" control operation. The info_pointer must point to a bit(72) variable containing the information to be forwarded to the FNP.

Example of a Control Tables Module

The source of a sample control tables module, sim_tables, is reproduced below. This module uses a line type of SYNC1, so the initial label is s1star.

This example implements a very simple protocol, and does not make use of all of the features described in this section; for examples of more sophisticated protocols, see the source of bsc_tables and polled_vip_tables.

Following the source of sim_tables is a line-by-line explanation of the purpose of each macro.

NOTES: Comment lines in 355MAP are indicated by either a star (*) in column 1 or the rem pseudo-operation in the operations field. When rem is used (as is usually the case), the string rem is replaced by blanks in the listing. To improve readability, frequent use is made of otherwise blank rem lines. In the example below, these lines are replaced by blank lines. Text following white space after the operand field is also treated as a comment. The /* and */ delimiters around the comments are entirely conventional; they are not required by the assembler.

It has been necessary for printing purposes to break some of the long control strings into two or more lines. In an actual source program these lines would not be broken unless they extended past column 72.

```
 1              lbl       ,sim_tables
 2              ttl       sim_tables -- to control sync lines
 3              pcc       off
 4              pmc       off
 5  * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 6  *
 7  *           sim_tables
 8  *
 9  *           these tables implement the protocol
10  *           of the sync line simulator. this simulator
11  *           is based on the vip7700 protocol.
12  *
13  *
14  *
15  *
16  *
17  *
18  * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
19
20  sim         null
21
22              symdef    sim
23              symdef    s1star
24
25              symref    begin
26              symref    error
27              symref    hanga
28
29              start     sim
30
31              pmc       save,on
32              tib
33              csbits
34              tconst
```

```
35                pmc        restore
36                ttls       constants for sync line tables
37   chr100       bool       100
38   chr140       bool       140
39   chr141       bool       141
40   chr124       bool       124
41   chr146       bool       146
42
43   iscn6        chstr      (rescan,search,soh,ignore,strlrc,
                              search,etx,ignore,cmplrc)
44   iscn7        chstr      (rescan,search,soh,ignore,
                              ignore,match,ack)
45   iscn8        chstr      (rescan,search,soh,ignore,
                              ignore,match,dle)
46
47   oscn1        chstr      (rescan,search,etx)
48   oscn2        chstr      (rescan,search,soh,ignore,strlrc,
                              search,etx,ignore,outlrc)
49   oscn3        chstr      (rescan,chktrm,etx)
50
51   ackmsg       chstr      (syn,syn,syn,syn,soh,chr141,ack,
                              stx,etx,chr146,eot,seteom)
52   nakmsg       chstr      (syn,syn,syn,syn,soh,chr141,nak,
                              stx,chr100,chr141,etx,chr124,
                              eot,seteom)
53
54   idlmsg       chstr      (syn,syn,syn,syn,soh,chr100,dle,
                              chr100,etx,seteom)
55
56                ttls       dial up control for simulator
57   s1star       tstflg     tflisn,0,slisn /* listen to line? */
58                wait       0,0,begin
59
60   slisn        contrl     sdtr+srts+stat  /* bring up leads */
61
62                wait       0,0,tshang  /* dial or hang? */
63                status     cd+cts+dsr,0,sdiald
64
65   sdiald       signal     dialup
66                goto       sget       /* start off protocol */
67                ttls       sget and srcvd input for sync lines
68   sget         tstwrt     sendr
69                tstflg     tfhang,0,shang
70
71                contrl     srec+rxmit
72                setime     0
73                wait       0,sendr,tshang
74                status     brkchr,0,srcvd
75                status     parity,0,nakit
76                status     exh,0,pause
77                status     xte,0,pause
78                status     0,dsr+cd,shang
79
80   srcvd        inscan     iscn6,nakit
81                inscan     iscn8,notidl
```

```
82                  dumpin
83                  goto        sntack
84
85   notidl         sendin
86                  dmpout
87
88                  bldmsg      ackmsg,error
89
90   sndmsg         holdot
91                  dcwlst
92                  cmd         sxmit
93                  output      (outmsg)
94                  cmd         sterm+rxmit
95
96                  wait        0,0,0
97                  status      term,0,sntack
98                  status      0,dsr+cd,shang
99
100  sntack         setime      0
101                 tstflg      tfhang,0,shang
102
103                 wait        0,0,tshang
104                 status      brkchr,0,srcvd
105                 status      parity,0,nakit
106                 status      exh,0,pause
107                 status      xte,0,pause
108                 status      0,dsr+cd,shang
109
110  pause          stpchn
111                 dumpin
112                 dmpout
113                 setime      1
114                 wait        nakit1,0,tshang
115                 status      0,dsr+cd,shang
116
117  nakit          stpchn
118                 dumpin
119                 dmpout
120
121  nakit1         tcntr       2000,rstcnt
122                 acntr       1
123                 contrl      srec
124
125  nakit2         bldmsg      nakmsg,error
126                 goto        sndmsg
127
128  rstcnt         scntr       0
129                 goto        nakit1
130  tshang         tstflg      tfhang,0,shang
131                 waitm
132
133  shang          stpchn
134                 dumpin
135                 dmpout
136                 goto        hanga
```

```
137              ttls       sendr for sync lines
138     sendr    outscn     oscn3,gtmore
139              goto       sendit
140
141     gtmore   signal     sndout
142              wait       0,sendr,tshang
143
144     sendit   holdot
145
146     resend   dcwlst
147              cmd        sxmit+srec
148              output     (outmsg)
149              cmd        rxmit
150
151     sndone   setime     30
152              tstflg     tfhang,0,shang
153
154              wait       badack,0,tshang
155              status     brkchr,0,gotack
156              status     parity,0,badack
157              status     exh,0,badack
158
159     badack   stpchn
160              dumpin
161              goto       resend
162
163     gotack   inscan     iscn6,badack
164              inscan     iscn7,badack
165              dumpin
166              dmpout
167              setime     10
168              contrl     rrec
169              wait       sndidl,sendr,tshang
170
171     sndidl   bldmsg     idlmsg,error
172              outscn     oscn2,error
173              holdot
174
175              dcwlst
176              cmd        sxmit
177              output     (outmsg)
178              cmd        sterm+rxmit
179
180              setime     0
181              wait       0,0,0
182              status     term,0,sntidl
183
184     sntidl   dmpout
185              tstflg     tfhang,0,shang
186              tstwrt     sendr
187              wait       0,sendr,tshang
188              end
```

Line	Purpose
1 - 4	Listing controls
5 -18	Introductory comments explaining purpose of module
22-23	Definitions of symbols referenced by other modules
25-27	Indicates that the indicated symbols are defined in other modules
29-36	Macros defining useful symbols
37-41	Assign symbolic names to some useful character values
43-49	Scan control strings
51-54	Canned message formats used by bldmsg op blocks
57	See if CS wants to communicate with channel
58	If not, wait until it does, go to control_tables label "begin" for preliminary testing
61-62	Wait here for dataset leads CD, CTS, DSR to come on. When they do, goto sdiald.
65-66	Tell CS about connection to communications device, and start looking for data at sget.
68	Check for output from CS for communications device.
69	Check for request from CS to disconnect.
71-79	Otherwise, wait for output from CS (go to sendr when it comes) or input from device (goto srcvd). Also watch for device errors such as parity or dropped dataset leads.
80	Scan input looking for LRC error, goto nakit.
81-83	Check for data message. If actual data go to notidl. Otherwise throw away input message and wait at sntack.
85-88	Send data to CS, throw out last ACK, build new ACK.
90-94	Hold current message for possible retransmission; set transmit mode and send message.
96-98	Wait for message transmission to complete or data set to drop (preferably the former).

Line	Purpose
100-101	Sent the ACK, test for CS request to disconnect.
103-108	Wait for next input message. goto srcvd when it arrives. parity error, NAK message. overrun, goto pause.
110-115	Wait a second and send NAK.
117-123	Throw away the input and output; count the bad message, testing for overflow first.
125-126	Construct a NAK message.
128-129	Reset counter to zero.
130	If CS requested disconnect, do it here.
133-136	Throw away the input and output, go to standard disconnect routine.
138-139	Make sure output ends in ETX; if not, go to gtmore, else go to sendit.
141-142	Ask for more output.
144	Hold the message for possible retransmission.
146-149	Send the message.
151-157	Wait 30 seconds for a response.
159-161	Bad ACK, retransmit message.
163-164	Check the LRC (logical redundancy check), look for ACK; if bad, transmit message again.
165-169	Discard the output, it was sent ok; wait 10 seconds for more to send, else send idle message.
171-173	Construct the idle message, compute LRC.
175-178	Send it.
180-195	Wait for idle to transmit, then throw it away.
186-188	Wait forever for more output from CS, or disconnect request.

SECTION 13

FNP HARDWARE MANAGERS

This section describes the FNP software that controls the four principal peripheral devices connected to the FNP: the Direct Interface Adapter (DIA), the Low Speed Line Adapter (LSLA), the High Speed Line Adapter (HSLA), and the console.

DIA

The Direct Interface Adapter (DIA) is the hardware interface between the FNP and the CS IOM. The FNP module dia_man is responsible for handling DIA interrupts, initiating DIA I/O, and passing on data sent from the CS to the other modules in the FNP.

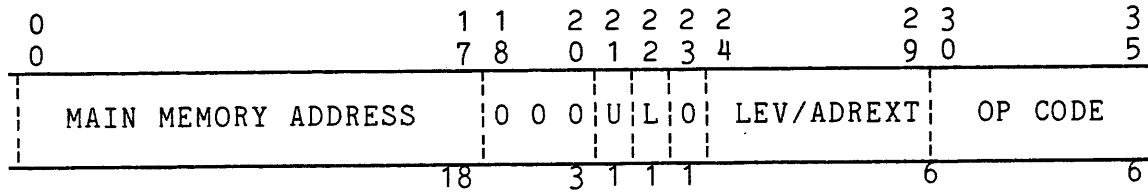
While the FNP is running, all data transfer over the DIA is initiated by dia_man; the only DIA operation that may be initiated by the CS is "Interrupt FNP", which is used to inform dia_man that a mailbox is ready to read.

Operation of the DIA

The address field of the CIOC instruction used to start DIA I/O points to the DIA PCW mailbox (at location 454 (8) in FNP memory), which contains a PCW that must be refreshed before each connect. The only part of the PCW that is interpreted is the address field, which points to a "list ICW" which in turn points to a list of DCWs that specify the actual I/O operations to be performed by the DIA. The tally field of the list ICW specifies the number of 36-bit words in the DCW list; each DCW actually consists of a pair of 36-bit words.

The format of a DIA DCW is shown below:

First Word Pair



Second Word Pair

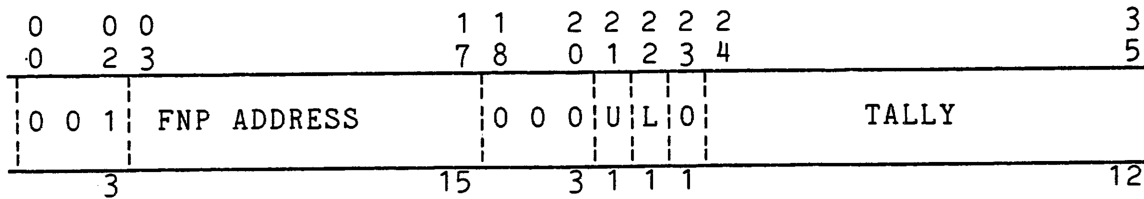


Figure 13-1. FNP DIA DCW Format

The DIA opcodes used by Multics Communication System are:

"transfer gate" (65) - i.e., read and clear CS memory, OR result to FNP memory

disconnect and interrupt FNP (70)

interrupt CS (73)

data transfer from FNP to CS (75)

data transfer from CS to FNP (76)

If the opcode is "interrupt CS", bits 24-29 of the first 36-bit word of the DCW specify the interrupt cell to be set; if the opcode specifies data transfer, these bits are the high-order 6 bits of the CS address.

All address fields in the DIA PCW, list ICW, and DCWs must specify 36-bit addressing; all tallies are in 36-bit words.

The dia_man module builds its DCW list in a reserved location in dia_man itself; space is reserved for a DCW list to transfer up to the maximum number of buffers that may be transmitted at once.

DIA Transactions

A complete transaction between the FNP and the CS generally requires more than one DIA I/O operation (e.g., reading a submailbox, transferring data, writing a submailbox, etc.). Accordingly, a "transaction control word" is used to determine the current state of the current transaction; it is examined whenever an I/O operation completes, and interpreted by the DIA transaction processor, dtrans. This word, which is kept in the internal storage of dia_man, can have any of the following values:

0	first interrupt since bootload
1	output pseudo-DCW list was read
2	output data transfer complete
3	submailbox was read from CS
4	buffered input data transfer complete
5	"blast" output message was read
6	data was read for patching FNP memory
7	submailbox containing input was written to CS
8	not used
9	submailbox was freed
10	submailbox was written to CS

A value of 1 through 7 means that further action by dia_man is required; any other value indicates that the current transaction is complete.

To prevent confusion and overcomplication in dia_man, only one transaction control word, and only one area for the storage of submailbox contents, are reserved in dia_man; thus, only one transaction can be going on at any time. This is enforced by means of a software lock (the "DIA lock") which is locked when a transaction begins and unlocked when it completes.

The initiation of a transaction can be requested by either the FNP or the CS at any time; such a request is processed by adding an entry to an appropriate queue and calling the secondary dispatcher (see Section 11) to schedule dgetwk, the routine that initiates DIA transactions. When dgetwk runs, it checks the DIA lock; if it is locked, nothing is done (dgetwk runs again later after the lock is unlocked). If the lock is not locked, dgetwk locks it and checks the mailbox queue and the DIA request queues (see below) to see if a transaction request is pending; if so, it initiates the transaction, as described later in this section. If no transactions have been requested, dgetwk unlocks the DIA lock and returns.

Queues

Two queuing mechanisms are used to drive DIA transactions: the mailbox queue and the DIA request queues.

The mailbox queue identifies the submailboxes that the CS wants the FNP to read. It is a circular queue of 16 entries; each entry contains either the number of a submailbox (0-15) or -1 to indicate a free entry. An entry is added to the queue whenever a mailbox interrupt is received from the CS; the oldest entry is removed by dgetwk when it finds the queue nonempty and builds a DCW list to read the submailbox specified by that entry.

As explained in Section 4, there are actually only 12 submailboxes, of which 4 (submailboxes 8-11) are controlled by dia_man. Submailbox numbers 12-15 are used by the CS to indicate that it has processed one of submailboxes 8-11 without modifying it.

The group of queues known collectively as the DIA request queues contains one queue for each configured channel and one global queue used to report errors to the CS. The queue address of each channel is kept in the TIB table entry for that channel (a zero value means that there are no queue entries for the channel). An entry is added to the queue of a channel by the denq subroutine when the FNP has information to pass to the CS, usually as the result of a "signal" op block in a control tables module (see Section 12). Each entry in the queue contains a submailbox operation code. Entries are extracted from the queue by the fetch subroutine when a submailbox containing an RCD (read control data) command is constructed by dia_man, as explained in more detail below.

Interrupt Handlers

Two routines handle DIA interrupts: dterm and dmail. Terminate interrupts, which are generated as a result of the "disconnect and interrupt" DCW placed at the end of every DIA DCW list constructed by dia_man, indicate the completion of a DIA I/O operation and are directed to dterm. This routine checks the DIA status to ensure that the I/O completed successfully, and if so it examines the transaction control word. If the value of the transaction control word is one that requires action (see above), dtrans is scheduled to proceed with the transaction; otherwise, the current transaction is complete, and dterm unlocks the DIA lock and calls gate, the subroutine that schedules dgetwk.

When the CS connects to the DIA, the result is a "special" or mailbox interrupt at the level corresponding to a submailbox that the FNP is supposed to read. This interrupt is handled by dmail, which copies the interrupt level (i.e., the submailbox number) into the next available entry in the mailbox queue and calls gate. When dgetwk runs, it finds the entry in the mailbox queue and initiates the reading of the submailbox as described above.

If the interrupt level is in the range 12-15, no submailbox is read; instead, the submailbox whose number is four less than the interrupt level (i.e., one of submailboxes 8-11) is marked free for further use by dia_man.

Summaries of DIA Transactions

TRANSACTIONS INITIATED BY THE CS

When the CS has control information or output data to send to the FNP, it fills in a submailbox as described in Section 4 and sends an interrupt over the DIA. This interrupt is handled by dmail as described above; when the submailbox is read, the transaction control word is set to "submailbox read" so that when the I/O completes and dtrans runs, the mailbox decoder (decmbx) is called. The I/O command in the submailbox is either WCD (for control information) or WTX (for output data). If it is WCD, decmbx dispatches according to a table of operation codes and takes the appropriate action. In many cases, this consists of setting a flag in the TIB and calling itest, the "test-state" entry of the interpreter. In a few cases, the operation requires further DIA I/O, but usually all that remains to be done is to "free" the submailbox by turning on the corresponding bit in the mailbox terminate interrupt multiplex word (see Section 4) and set the transaction control word accordingly. When the I/O to update the TIMW terminates, the transaction is complete.

If the I/O command is WTX, the submailbox contains the address and length of a "pseudo-DCW" list containing the addresses and tallies of data buffers in tty_buf. In this case, dia_man connects to a DCW list to read them into a reserved area in dia_man. When the I/O completes and dtrans runs, another DCW list is built making use of the information in the pseudo-DCWs, buffers are allocated for the actual output, and a connect is done to copy it. When this I/O operation completes, dtrans calls the write subroutine, which checks to see if output is currently being sent to the channel, as indicated by the TIB flag tfwrit. If the flag is on, dia_man simply extends the output chain pointed to by t.ocur to include the new output. If the length of the resulting chain is no greater than a per-channel threshold (usually 2 or 4 buffers depending on channel speed), a flag is set in the submailbox to indicate a "send_output" operation; in this case, the submailbox is rewritten to the CS at the end of the transaction. If the tfwrit flag is off, then the new output is appended to the chain pointed to by t.ocp (or t.ocp is set to point to the new output if there is no existing chain) and the iwrite entry of the interpreter is called so that the control tables can deal with the output. Then the mailbox is finally freed or rewritten, completing the transaction.

If there is insufficient free buffer space for either the pseudo-DCWs or the output itself, dia_man frees any buffers allocated so far in connection with the output transaction, and reschedules processing of the submailbox. This is done by scheduling the rpmbx subroutine to run after six seconds; this routine adds an entry to the mailbox queue for the specified submailbox. As a result, dgetwk rereads the mailbox exactly as if an interrupt for it had arrived from the CS, and the entire transaction is retried.

A WCD submailbox containing a blast operation code is a special case of output data, in that the submailbox contains the address of the actual output. In this case a DCW list is set up to read the blast message, after which the data is copied and sent to every dialed-up channel.

TRANSACTIONS INITIATED BY THE FNP

When denq is called (either as a result of a signal or sendin op block, or by write, hsla_man, or lsla_man to request more output), an entry is added to the request queue for the specified TIB containing the specified mailbox operation code. The count of pending queue entries is increased by one, unless there is already an entry in the queue of that channel containing an accept input operation code; an accept input blocks the queue, for reasons explained below. Similarly, a call to derrq adds an entry to the global queue. In either case, when dgetwk runs, it sees that the queue entry count has been incremented, and calls the filmbx subroutine, which allocates an FNP-controlled

submailbox, fills it in with an RCD I/O command, and calls the fetch subroutine to get an entry from the appropriate request queue; fetch checks the global queue first and returns the oldest entry in this queue if it is not empty. Otherwise, the request queue for the channel whose TIB list entry is addressed by the current request queue pointer is examined, and if it contains a pending request, this is returned; if not, the request queue pointer is advanced and the queue of the next channel is checked. Once an entry is found, the request queue pointer is set to point to the next entry in the TIB list, so that the request queues are examined in a round-robin fashion, thus giving more or less equal service to all active channels. If the entry contains any operation code other than accept input, the entry is freed by setting it to zero, and the queue buffer containing it is freed if there are no more entries in it.

The operation code is put in the FNP copy of the submailbox, and the line number of the channel and any additional information is filled in. If the operation code is "line disconnected", indicating a hangup condition, certain fields in the TIB are reinitialized, and any associated echo and/or CCT buffers are freed. A DCW list is built to write the submailbox to the CS and to turn on the corresponding TIMW bit; when the CS has processed the submailbox, it interrupts the FNP on the corresponding level (as explained above), thus completing the transaction.

INPUT DATA

The transmission of input data is a two-stage operation, and accordingly is somewhat more complicated. When a sendin op block is executed, the interpreter calls denq with an accept input operation code; denq copies the current input chain (pointed to by t.icp) into the DIA input chain (pointed to by t.dcp); if it is longer than 10 buffers or more than 600 characters, it is split up into two or more messages by marking the last buffer of each "submessage" with a "last" flag and putting an additional accept input entry in the queue. As a rule, each accept input entry must correspond to a complete input message ending with a buffer containing a "last" flag.

When filmbx processes the queue entry containing the accept input, it calls incnt to count the number of characters in the input message addressed by t.dcp; this count is placed in the submailbox. It may be that there is insufficient space in tty_buf for the input message, in which case the accept input has to be retried later; therefore the entry cannot be removed from the queue until the input has been accepted. That is why entries added to a queue behind an accept input entry do not increase the queue entry count immediately; they must not be processed until the accept input is removed from the queue. When the accept

input is copied to the submailbox, the queue is marked with an "active" flag so that it will not be picked up for another submailbox.

If the data message is longer than 100 characters, the accept input submailbox is sent to the CS to tell it how many characters to accept. If the CS has room for the input, it modifies the submailbox to describe where to put it in the circular buffer as described in Section 4; the I/O command is changed to RTX to tell dia_man that the input should be sent. The addresses and tallies in the submailbox are used by indata to build a DCW list to copy the input to the CS. At this point the accept input entry is removed from the request queue, and the queue entry count is increased by the number of entries (if any) behind it in that queue, up to and including the next accept input in the queue of that channel (if there is one).

When the I/O to copy the input completes, dtrans frees the DIA input chain and connects to a DCW list to free the submailbox, completing the transaction.

If the input message is 100 characters or less, it is copied directly into the submailbox, and the submailbox is written to the CS. The transaction control word in this case is not set to "mailbox written", but rather to "input sent in mailbox". This indicates to dterm that it is neither to unlock the DIA nor to schedule dtrans; the CS is obligated to respond with an interrupt to free or update the submailbox. When this interrupt arrives, dmail ensures that either dtrans is scheduled or the lock is unlocked.

If the CS has room for the input, it sends an interrupt at the appropriate mailbox-freeing level; dtrans then frees the input chain, removes the accept input request from the request queue, and updates the queue entry count as described above.

If the CS does not have enough space to accept the input (whether short or long), instead of an RTX command, it updates the submailbox with a WCD command and a "reject request" operation code. When this submailbox is read, the accept input queue entry is marked with a "rejected" flag, and the retry routine, dretry, is scheduled to run one second later. When dretry runs, it finds the rejected request in the queue, turns off the "rejected" and "active" flags, and increments the queue entry count, thus restarting the transaction.

Quits and hangups override retrying a rejected request. This is enforced by the following mechanism: when a quit or hangup entry is added to a request queue that already contains an

accept input entry, the accept input entry is marked with a "quit" flag; if an accept input entry with a "quit" flag is rejected, it is not retried, but rather it and all subsequent accept input entries are cleansed from the queue by the cleanq subroutine. In addition, if a quit or hangup entry is added to a queue containing an already rejected accept input entry, the queue is cleansed then and there. In this case, dretry finds that the request to be retried is gone, and takes no action.

LSLA

The Low Speed Line Adapter (LSLA) is the interface between the FNP and asynchronous communications channels with speeds of from 110 to 300 baud. Up to six LSLAs can be configured per FNP. They are managed by the lsla_man module.

Operation of the LSLA

Once it is started by init (see Section 15), the LSLA runs continuously. It provides one input "frame" every 100 milliseconds, and expects one output frame every 100 milliseconds. Each such frame consists of 60±1 one-character "time slots", of which 52 contain data characters received from or to be sent to the various channels. A 10 character-per-second (110 baud) channel uses one slot per frame; a 15 cps (133 or 150 baud) channel uses two slots per frame, but the second slot is unused in every other frame; a 30 cps (300 baud) channel uses three slots per frame. The slots and channels are bound together during FNP initialization; the LSLA table entry for each slot indicates the channel speed and sequential position of the slot within the channel, and contains the address of the TIB of that channel. This table is described in Section 10.

Because of slight variations in terminal speed, output frames may actually be 61 characters long, and input frames 59. The first five characters in a frame must be four SYN characters and an STX; the sixth is an unused T & D slot, and the data slots are in character positions 7-58. The ICWs used by the LSLA are set to start a frame at the fourth character position (right half of the second word) in a 32-word buffer, so that the longest (61-character) frames are right-adjusted in their buffers. The first word of the buffer is used for control information.

An odd-parity US character (037) is used as a fill character; its presence in a slot indicates that no character is to be sent to, or has been received from, the channel. An odd-parity ESC character (233) indicates that the next nonfill character associated with the particular channel (either in the next slot or the next frame) is a command to the LSLA (in an output frame) or a status character (in an input frame). Status and command characters have 8-bit odd parity; data characters have 8-bit even parity.

Whenever a complete input frame is ready to process, and whenever a complete output frame has been sent, the LSLA stores status in the software communications region and interrupts the FNP.

Interrupt Processor

The LSLA interrupt processor, lip, is invoked to handle an LSLA interrupt at the end of a frame. It examines each pending status word in the software communications region. When it handles a status, it first checks to make sure that the status indicates that the LSLA is running normally; if it is not, it determines the nature of the error, and, if appropriate, queues an error message to be sent to the CS and/or reconnects to the LSLA. Some extra tests are made to see if the LSLA has just been started, since it may take a few frames to begin running properly.

TRANSMIT STATUS HANDLING

Two static buffers following the software communications region are used alternately for output frames. The two output ICWs in the hardware communications region are initialized to point to these two buffers. When the first one is exhausted, the LSLA sends an interrupt and starts outputting the second one. The interrupt processor, meanwhile, refreshes the first ICW, initializes the first buffer with fill characters, and schedules the output frame generator, loutpt, to fill in the slots in the first buffer with appropriate data characters. When the LSLA finishes outputting the second buffer, it sends an interrupt and switches back to the first one; lip refreshes the second ICW and the second buffer as above.

RECEIVE STATUS HANDLING

Two buffers are initially allocated for input frames; when the first one is filled, the LSLA stores status, interrupts the FNP, and starts on the second one. The interrupt processor must refresh the first ICW and provide a buffer to switch to when the second one is filled. First it ensures that the LSLA is properly synchronized by checking that the fifth character in the frame is really an STX; if it is not, the frame is not processed, and the same buffer is used for the next input frame. The software communications region flag sffnsx is turned on to indicate that the last frame was incorrect; if the STX is missing or misplaced in two successive frames, the flag sffrsy is set, and the next time transmit status is processed a "resynchronize" PCW is sent to the LSLA to get it back in sync.

When receive status is accompanied by a correct input frame, the input processor checks to see if it consists entirely of fill characters. If it does, the buffer containing the frame can be reused, so the most recently exhausted ICW is set to point to it. If the frame contained characters other than the fill character, the input frame processor, linput, is scheduled to interpret the frame; a new buffer is allocated, and the most recently exhausted receive ICW is set to point to the new buffer so that the LSLA can start to fill it when the frame currently being received is complete.

ABNORMAL STATUS HANDLING

If a status other than a normal pretally runout is received, special action may be necessary. If any of the normal dataset leads of the LSLA have dropped, the LSLA must be resynchronized, so sffrsy is turned on. If tally runout status occurs, it signifies that an ICW did not get refreshed quite fast enough; lip simply ensures that both applicable ICWs (either send or receive) are properly filled in and reconnects to the LSLA. Other unexpected status conditions are reported to the CS via the error message mechanism, and the LSLA is resynchronized if necessary.

Output Frame Generator

The LSLA output frame generator, loutpt, is scheduled to fill an output frame every 100 milliseconds. (The SYN and STX characters are assembled into the output frames and do not need to be refreshed.) For each slot, it checks the LSLA table entry to see what is going on for the associated channel. If a dcwlst op block has been executed for the channel since the previous output frame was processed, ltfdcw is on, and the subroutine loudcw is called to process the first sub-op in the list. If this is a command sub-op, it may result in a command sequence being started; an odd-parity ESC has been placed in the slot, and ltfesc turned on so that loutpt will put a command character in the next slot for the same channel. If the sub-op is output, the outprc subroutine (described in Section 14) is called to set up the output chain and put the channel in transmit mode. Other sub-ops are handled by the input frame processor.

If no control sequence is being sent, loutpt must decide whether to place an output character from the CS or an input character from the echo buffer in the slot. If the channel is not in transmit mode, and there is a pending character in the echo buffer, the echoed character is put in the slot. If the channel is in transmit mode, the decision as to whether to use an output character or an echo character depends on the setting of the TIB flag tfecho, which is used to prevent output and echo characters from being interspersed. If the flag is on, it indicates that an echo sequence is going on, and echo characters have priority; otherwise, output characters have priority and echoing is not resumed until the output chain is exhausted. If tfecho is on, but there are no pending echo characters, the flag is turned off and an output character is sent. If there are no output or echo characters pending, the slot is left as it is (containing a fill character).

When an output character is put in a slot, the move subroutine is called to adjust t.pos to reflect the new column position of the terminal. The character is given appropriate parity for the type of terminal, and copied into the output slot. If it was the last character in a data buffer, the buffer is freed and the output chain adjusted accordingly; if the length of the chain crosses the output threshold for the channel, a request for more output is queued to be sent to the CS. If the output chain is now completely exhausted, tfwrit is turned off so that dia_man can know to call iwrite the next time output for the channel arrives from the CS.

Command Sub-ops

A command (cmd) sub-op of a dcwlst op block in the control tables usually requires the sending of a command sequence to the LSLA. These sub-ops are handled by the comand subroutine, which constructs a command character based on the controls specified in the sub-op. If one of the controls is a request for status, a special status request command character must be sent after the normal command character (if any). Of course, each of these characters must be preceded by odd-parity ESC characters.

A line break sent to the channel causes the line to drop for up to 600 milliseconds; accordingly, a command to generate a line break must be followed by 600 milliseconds worth of fill characters. The ltfbrk flag is set to indicate to loutpt that a line break is in progress, and t.bcnt reflects the number of character times to wait before attempting to send data to the terminal. Since the first character after a break is sometimes garbled, the first character sent by loutpt after a line break is a DEL character.

It should be realized that by the time loutpt runs, the control tables have reached a wait state; it is possible for an event occurring between output frames to start the control tables running again, with the result that a new dcwlst op block is to be started. On the other hand, once the escape character has been sent, the following command character must be sent in the next slot for the channel. Accordingly, ltfesc takes priority in loutpt, followed by ltfdcw, the flag indicating that a new dcwlst op block has been encountered.

Input Frame Processor

The input frame processor, linput, is scheduled whenever a complete input frame has been received. Its task is to examine every data slot in the frame and take appropriate action. Any slot containing a fill character can be ignored, as can any slot not corresponding to a configured channel. If the slot contains an odd-parity ESC character, the LSLA table flag ltfste is set so that the next slot for the same channel will be treated as a status character. A status character is converted to an appropriate status word recognizable by the control tables, and the interpreter is called. A line break condition is bounded by two status characters, one with the "line break" bit on, and one with this bit off; all intervening slots for the same channel are ignored.

If the slot contains a data character, its treatment depends on whether or not an input or rdtly sub-op is being processed. For an input sub-op, the character is compared against the character specified in the sub-op; if they are equal, the sub-op is satisfied and t.dcwl and t.dcwa are updated accordingly. Characters appearing during a rdtly sub-op are treated like ordinary data characters, but the associated tally is decremented for each character found. Data characters are ignored if the channel is not in receive mode.

If the slot contains a normal input character, the move subroutine is called to determine if it is a carriage-movement or case-shift character, and to adjust t.pos to reflect changes in the column position of the terminal. If the channel is in crecho, lfecho, or tabecho mode, and the character is one of the relevant characters, appropriate character(s) are added to the echo buffer. For terminals using case-shift characters, uppercase characters are marked by turning on the 100(8) bit. In echoplex mode, all characters are added to the echo buffer.

If an input chain does not already exist for the channel, a buffer is allocated and its address stored in t.icp. Otherwise the new character is added to the last buffer in the current chain, if possible; if this buffer is full, a new one is allocated. If the allocation results in a chain of maximum allowable length, exhaust status is sent to the control tables, which may take whatever action is deemed appropriate (for example, taking the channel out of receive mode). This mechanism is intended to prevent any one channel from absorbing an excessive portion of the available buffer space.

The input character is then looked up in the break list to see if break character status should be sent to the control tables. If it is, then tfwrit is turned on so that if the channel is in echoplex mode and currently receiving output, the completed input message will be echoed when the current output is complete, and before any additional output from the CS is sent to the channel, since dia_man will not append the latter to the current output chain.

If the channel is in blk_xfer mode (indicated by the TIB flag tffrmi) no break characters are recognized while a "frame" or block of input is in progress, except for the frame-ending character. The TIB flag tffip indicates whether a frame is in progress. This flag is turned on by the appearance of a frame-begin character while tffrmi is on, and turned off by the appearance of a frame-end character.

Echoing

The puteco subroutine is used to add a character to the echo buffer for later echoing. The geteco subroutine is called by loutpt to obtain a character from the echo buffer. The echo buffer is circular, with an input pointer and an output pointer maintained by puteco and geteco respectively.

When geteco returns a character, it normally advances the output pointer. There are two exceptions, however. In tabecho mode, when geteco finds a tab in the echo buffer it replaces it with the negative of the number of spaces to echo; successive calls to geteco result in its returning a space and incrementing this count, only advancing the output pointer when the count reaches zero. Similarly, when a carriage-movement character requiring delays is found, it is returned but replaced in the echo buffer by 128 plus the number of delays required. Thus when geteco encounters a character whose 200(8) bit is on, it returns a NUL and decrements the character value by one; when this reaches 128, the output pointer is updated.

HSLA

The hsla_man module is responsible for operation of the High Speed Line Adapter (HSLA) and all of the subchannels connected to it. The module is logically divided in two sections, the call side and the status processing side.

The HSLA manager uses four data bases to control the operation of the HSLA: the hardware communications region, used by the hardware for indirect control words (ICWs), etc.; the HSLA table, used during initialization to configure the subchannel; the software communications region, used to keep HSLA-specific data such as status and ICW pointers; and the terminal information block (TIB), the database of the control tables. The format of the hardware communications region and HSLA control words may be found in Section 10 and the Formats PLM, respectively. The HSLA software communications region contains the hardware status queue, pointed to by the status ICW of the hardware communications region, and the software status queue, pointed to by two pointers in the software communications region. (The format of the software communications region is described in Section 10.) The hardware status queue is filled by the hardware via the status ICW, and for each status word deposited an HSLA interrupt occurs. These interrupts are processed by the hintr routine, which removes status words from the hardware status queue and places them in the next available spot in the software status queue. The software communications region also contains flag bits used to control the HSLA and pointers to the buffers corresponding to the ICWs in the hardware communications region.

The TIB contains pointers to the current input and output chains, the current DCW list from the control tables, if any, and status and flag bits.

Calls to hsla_man

The call side of hsla_man contains four entry points: hdcw, hcfg, hmode, and hgeti. The hdcw entry is the DCW list processor, called by the interpreter to process standard DCW lists found in the control tables. The hcfg entry is the HSLA configuration change entry, called by the interpreter to process configuration change sub-ops. The hmode entry is used when the echoing mode bits are changed to signal the software that a new character control table (CCT) may be required. The hgeti entry is used for the replay and polite mode operations, to test for the presence and make copies of any partial input line.

DCW list processing consists of interpreting each sub-op in the DCW list. Command sub-ops are processed by the cmdprc subroutine, and if receive mode is turned on, the bldibf subroutine is called. This subroutine sets up the input buffers for the channel; it also checks for a partially filled input buffer and sets up the ICWs required to complete the input buffer. This is required if the input is completed in another buffer. The dia_man routine is not able to transfer the data as a contiguous character stream, as all data must be on a 36-bit boundary. It should also be noted that hsla_man always sets the tally of input buffers to one 36-bit word less than the maximum that would fit, to allow for possible input tally runouts that may store one character of data beyond the maximum tally specified.

If the processing of a command sub-op leaves the channel in transmit mode, the bldobf subroutine is called. This subroutine calls the outprc utility routine to process an output sub-op, and then the seticw subroutine, to set up the output ICWs. The processing of the DCW list is suspended upon encountering an output sub-op, to allow the data transfer to complete. When it does, hdcw is called to complete processing of the DCW list.

The processing of each piece of a DCW list is completed when hdcw issues an HSLA PCW to the channel. This is the mechanism used by hsla_man to indicate changes in modes and dataset leads, and to signal the software that these changes are complete. Therefore, in most cases, when hdcw issues the PCW, the PCW operation code is a "request receive status", which causes an interrupt to occur and status to be stored indicating the new state of the channel.

HSLA Status

All status is queued by hintr as described above and processed by the hstprc routine. This routine does some initial checking on the status and then determines whether the status is receive or transmit type and processes it accordingly. Actual status processing is done by subroutines that correspond to each important bit in the status and the routines are dispatched by status bit lookup tables. These tables list the important status bits in an order that allows the first routine dispatched to do all functions related to that status. Thus, only one routine is normally dispatched for each status word.

Besides dispatching the main status processing routines, hstprc is responsible for keeping the channel in step with the software and running properly. The largest part of this job is making sure that the ICW indicators in the status are in agreement with the software ICW indicators for both receive and transmit ICWs. If the software and hardware appear to be out of phase with respect to the ICW indicator, the software attempts to recover the channel and reestablish a consistent channel state. The hstprc routine also handles echoplex mode by running the echock subroutine to make sure that any available echo data is output. If certain conditions are met when hstprc finishes processing all queued status (e.g., last status was receive status, there is no active mode, etc.), hstprc calls hdcw to process any remaining DCWs.

Example of HSLA Processing

To clarify this discussion, an example of an ASCII line connected to the HSLA is useful. Assume that the channel is not dialed up, but that data terminal ready has been presented to the dataset. The phone rings and answers, presenting clear to send, carrier detect and dataset ready to the HSLA subchannel. The subchannel fabricates a status word with the dataset status change indicator on and the new dataset status. This status is stored via the status ICW in the hardware communications region, and an interrupt is generated for this subchannel. The hintr routine is called to pick up the status, place it in the software status queue, and schedule hstprc. The status is processed and the ipdss subroutine sends the new status bits to the control tables. The CS is informed of the dialup and responds with an output message. The control tables call hdcw with a DCW list to send the output. This DCW list consists of three sub-ops: a command sub-op to set transmit mode, an output sub-op to send the output, and another command sub-op to reset transmit mode and send terminate status. The DCW list processor, hdcw, processes the first two sub-ops, sets up the output ICWs, and issues a PCW to set transmit mode. This results in a status store of a receive status word. The status processor records the new state of the channel, notices that output is still going on and returns. The greeting message is usually two buffers, so at the

completion of the first one, an output pretally runout occurs and status is stored. The hstprc routine calls the opptro subroutine to free the output buffer and set up the ICW for the next buffer. Since there are only two buffers, no ICW is set up. Later another output pretally runout occurs and hstprc calls opptro again. Almost immediately an output tally runout occurs, indicating that all output is complete, and hstprc calls optro, which notices a remaining DCW list and calls hdcw. The hdcw routine issues a PCW turning off transmit mode; hstprc processes the status and sends terminate status to the control tables. The control tables are waiting for this status and proceed to issue a DCW to set receive mode. This DCW list is processed by hdcw, which sets up the input buffers, sets up the PCW to turn on receive mode and, since this is the first time the channel has been in receive mode, sets the address of the default character control table (CCT) in the base address word (BAW) of the hardware communications region. The CCT is used by the HSLA to determine what status to store, if any, for each character received. The PCW is then issued and the status is processed to update the channel state. The device begins to send characters, which are stored by the HSLA. If the size of the first input buffer is exceeded, the HSLA generates input pre-tally runout status, which causes hstprc to call ipptro; the buffer is placed on the input chain and a new buffer is allocated. When a newline (or any "break" character) is sent, the HSLA stores terminate status and switches ICWs. The hstprc routine calls ipterm, which places the buffer on the input chain, sends status to the control tables, and allocates a new input buffer. The control tables send this input to the CS and if any output results the cycle is repeated; otherwise, the channel remains in receive mode.

CCT Management

The hsla_man module is responsible for ensuring that the base address word (BAW) in the hardware communications region for each channel points to the correct CCT for that channel. Because the 6 low-order bits of the CCT address are not stored in the BAW, every CCT must begin at the 0 mod 64 address.

For a CCT that is coded into a control tables module and specified in a setcct op block (see Section 12), the CCT address is simply stored in the BAW and in the software communications region. This is the normal method for synchronous channels. In general, for asynchronous channels, the CCT is constructed according to certain modes as specified in the TIB flags; to save space, such CCTs are shared among channels with the same mode settings.

To enable the sharing of CCTs, each dynamic CCT is described by a CCT descriptor; these descriptors are chained together, and the first one in the chain is pointed to by .crcct in the system communications region. The format of a CCT descriptor is as follows:

Field	Description
cct.nx	address of next CCT descriptor
cct.pr	address of previous CCT descriptor
cct.ad	address of CCT
cct.sz	size of CCT in words
cct.rc	reference count (number of channels using this CCT)

When the state of one of the relevant modes (echoplex, tabecho, lfecho, breakall, or blk_xfer) changes, the subroutine makcct is called to construct an appropriate CCT; such a CCT would generate terminate (break character) status for newline and ETX characters, plus carriage returns if in lfecho mode (or if in breakall mode, for all characters); marker status for tabs (if in tabecho mode) or for all characters (if in echoplex mode); or to switch to a second CCT upon receipt of a frame-begin character in blk_xfer mode. The shrcct subroutine is then called to examine all currently-allocated dynamic CCTs to see if there is one that is identical to the newly-constructed one; if so, the reference count in that CCT's descriptor is incremented by one, and its address stored in the BAW and the software communications region. If no matching CCT is found, a new descriptor is allocated, and a block on a 64-word boundary is allocated and the new CCT copied into this block. The descriptor is threaded onto the head of the chain of descriptors. The reference count in the descriptor of the channel's old CCT (if any) is decremented by one; if it goes to zero, the CCT and its descriptor are freed.

Echoing

When marker or terminate status is generated, the scan subroutine is called to examine the contents of the current input buffer if any echoing modes (lfecho, crecho, tabecho, or echoplex) are on. Each character is looked up in the carriage movement table in the channel's device info table (see Section 12); if any echoing is required, the appropriate

characters are placed in the echo buffer. The scan subroutine also keeps track of the terminal's column position so as to be able to determine how many delay characters to echo along with a carriage return or tab, and how many spaces to echo for a tab in tabecho mode.

The echock subroutine checks to see if the echo buffer contains any unechoed characters, and, if so, sets up an ICW to output them and puts the channel in transmit mode. This subroutine is called by scan or at the completion of status processing if the channel is not currently in transmit mode; it is also called when output tally runout status is reported by the channel. This arrangement ensures that echoing will be performed as soon as possible without interfering with current output.

CONSOLE

The FNP console, if one is configured, is managed by the module console_man. This module is invoked either from other modules to print messages, or as a result of operator intervention. Messages are printed on the console by init if errors are detected during initialization, and by the fault handler when the FNP crashes. Operator input is accepted after the operator presses the "interrupt" button on the console.

The modules that call console_man to print messages do so by calling the wcon subroutine. When this routine is called, it is assumed that the FNP is not engaged in normal operation, but rather is either being initialized or is about to crash. Accordingly, once the connect has been done to print the message, wcon simply waits for a terminate interrupt; this interrupt is handled by contip, which returns immediately to the caller.

During normal operation, interrupts are handled by consol, which schedules either tmcon or spcon, depending on whether a terminate or a "special" interrupt was received. A special interrupt is generated when the operator presses the interrupt button on the console. Two routines are used to perform console I/O during normal operation: wrcon, which prints a message and then issues a read to the console, and write, which just prints a message. Each of these routines is passed the address of a routine to be scheduled when the I/O completes.

When the operator presses the interrupt button, a special interrupt occurs and spcon is scheduled. When spcon runs, it calls wrcon to print the message "???"; when tmcon handles the terminate interrupt at completion of the write, it transfers back into wrcon to issue a read. This read terminates when one of the following happens: a carriage return is typed, a "control-x" is typed, or a 30-second timer runs out. In the latter two cases, the write and read are reissued; otherwise the spconb routine is scheduled to process the input. If the input is a recognized command, the appropriate action is taken, after which wrcon is called again with a message of "MORE?" to allow further commands to be input. If the input is null (i.e., it consists of just a carriage return), console_man returns to the secondary dispatcher; no further console I/O is done until another special interrupt occurs. If an unrecognized command is typed, wrcon is called with the message "WHAT?"

Recognized commands are PEEK, ALTER, and ABORT. The PEEK command is used to display the contents of FNP memory, and takes one or two arguments: the first is the starting address to be displayed, and the second, if present, is the number of words to display (if the second argument is omitted, one word is displayed). Eight words are displayed on a line; the write routine is called for each line until the request has been satisfied.

The ALTER command is used to modify a word of FNP memory. It takes two arguments: the first is the address of the word to be modified, and the second is the value to be put at that address; after modifying the word, the alter subroutine calls the peek subroutine to display it.

Arguments to PEEK and ALTER are octal numbers, and are separated by commas and no spaces. These arguments are processed by the idx subroutine.

The ABORT command crashes the FNP by transferring to conabt, the entry in the utilities that simulates a fault called "console abort".

It should be noted that the console software prints uppercase output and expects uppercase input, and that the console channel interrupts on receipt of a carriage return, not a newline.

There is rarely any occasion to use any of the console commands; the effects of the PEEK and ALTER commands can be achieved much more easily and flexibly by means of the debug_fnp command (described in Appendix B).

SECTION 14

FNP UTILITY FUNCTIONS

This section describes various utility functions used in the course of operation of the FNP software. These functions include buffer space management, TIB address calculation, fault processing, metering, and output sub-op processing, all performed by routines in the utilities module; and the memory tracing facility, handled by the trace module.

SPACE MANAGEMENT

Buffers are allocated and freed either individually or in threaded lists. The getbuf, getubf, and frebuf subroutines are used to allocate and free a single buffer; getlbf and frelbf are used to allocate and free buffer chains. The getubf subroutine is used for noncritical buffers, and refuses to allocate a buffer if fewer than 20 32-word blocks are available; getbuf allocates a buffer whenever sufficient space is available. In addition, space for control blocks (as distinguished from data buffers) is allocated and freed by the subroutines getmem and fremem. The free pool consists of a chain of free blocks: the address of the first free block is in .crnxa in the system communications region, and the free blocks are chained together by means of forward pointers; see the description of a free block in Section 10. The free blocks are chained together in order by address; in other words, the forward pointer always points to a block with a higher address than the current block. When a block is allocated, getbuf or getmem adjusts the forward pointer in the immediately preceding free block. When a block is freed, frebuf or fremem checks to see if the newly-freed block is immediately preceded and/or followed by a free block; if so, the adjacent blocks are consolidated into one larger block.

The size of a buffer can be any multiple of 32 words (up to 256 words). When a buffer is allocated, a size code indicating the correct multiple of 32 words is placed in the high-order three bits of the second word (0 = 32 words, 1 = 64, etc.); the rest of the buffer is set to zero. All buffers are allocated starting at 0 mod 32 addresses. The size of a control block

allocated by getmem can be any even number of words and can begin at any even address.

When getlbf is called to allocate a chain of buffers, it calls getbuf repeatedly, setting the forward pointer (in the first word) of each buffer to the previously-allocated buffer. When frelbf is called to free a buffer chain, it calls frebuf for each individual buffer until it has freed one whose forward pointer is zero.

TIB ADDRESS CALCULATION

The gettib subroutine is called by various modules (notably dia_man) to obtain a TIB address given a 10-bit line number. This is done by first using the high-order portion of the line number to determine the adapter (HSLA or LSLA) on which the specified channel is configured, and then finding the entry for that adapter in the IOM table. The gettib routine uses the knowledge that the HSLAs are on IOM channels 6 through 8 and the LSLAs are on channels 9 through 14. The IOM table entry points to the HSLA or LSLA table for the relevant adapter; the low-order part of the line number is then used to find the correct entry within the HSLA or LSLA table, which in turn contains the TIB address. The formats of the various table entries are described in Section 10.

FAULT PROCESSING

The hardware fault vectors at absolute locations 440 through 447 point to an array of locations in the utilities module, starting at hfv. When a fault occurs, a tsy to the location specified by the fault vector is executed; at this location is another tsy to the fault processor, fp. Thus fp can determine the type of fault by seeing where it was called from (which location after hfv), and the location at which the fault occurred by examining the target of the original tsy.

The contents of the machine registers and the value of the instruction counter at the time of the fault are saved in a known location in the utilities module (whose address is kept in .crreg in the system communications region). For almost all faults, the FNP then crashes in an orderly fashion, as described below. Certain types of IOM channel fault, however, can be restarted; these are discussed later in this section.

When crashing the system, fp first determines the name of the fault based on the location at which the fault processor was entered; then it masks all the HSLAs and LSLAs and sets the interrupt vectors for all devices except the console to point to an "ignore" subroutine that simply restarts the interrupt. The

console terminate interrupt vector is set to point to contip (see the discussion of the FNP console in Section 13). A subroutine is then called to print a message on the console describing the fault. This message includes the name of the fault and the instruction counter; if the fault is an illegal opcode, the contents of the faulting instruction are printed as well. For an IOM channel fault, the channel number and the fault status are printed instead of the instruction counter. This subroutine also puts the necessary information in the location from which it is later sent to the CS.

A partially preset DIA DCW list is now completed in order to write the crash information into words 6 and 7 of the CS mailbox header; after allowing time for the I/O to complete, another DCW list is sent to the DIA to interrupt the CS at the "emergency interrupt" level. This interrupt is interpreted by the CS as described in Section 8.

Finally, mask PCWs are sent to all FNP I/O channels, all interrupts are disabled, and a dis instruction is executed, effectively stopping the FNP.

IOM Channel Faults

Two cases of IOM channel fault are considered nonfatal. One of these is a fault status of 14 (octal) on channel 0, which used to occur as a result of some hardware problems in the clock on the DATANET 355 (it might still occur at a site with an old 355 in which the relevant field change has never been made). The other case is a parity error on an HSLA channel. The fault processor checks for these two cases by examining the IOM fault status words starting at location 420. If the appropriate status is found, an error message is queued for dia_man to handle as described in Section 13; then the registers are restored and a transfer is made to the value of the instruction counter at the time of the fault. Note that the vector for IOM channel faults is set to return while the fault processor is running, so that further IOM channel faults cannot interfere with its attempts to recognize a nonfatal IOM channel fault.

METERING

Two metering entry points are provided in the utilities module: meterc, which updates "counting" meters, and metert, which updates "timing" meters. A call to meterc results in adding 1 to the specified counting meter; a call to metert adds a specified amount of time to a specified timing meter. Space is reserved for 50 meters of each type. At present, no timing meters have been defined, and counting meters are only used to keep track of certain abnormal conditions. The following counting meters have been defined:

1 LSLA output tally runout

2 LSLA input tally runout

3 abnormal LSLA status

4 quit signalled as a result of loss of carrier

5 no CCT available corresponding to a requested mode change

6 abnormal printer status

These meters are stored in a static area in the utilities module and can be inspected by using the display request to the debug_fnp command (see Appendix B).

In addition, some meters referring to faulty LSLA input frames are kept in the hardware communications region for each LSLA; see the description of the hardware communications region in Section 10. Idle time metering and instruction counter sampling are implemented as described in Section 11.

OUTPUT SUB-OPS

The outprc subroutine is called by both hsla_man and lsla_man to process an output sub-op of a dcwlst op block. The format and purpose of the output sub-op are described in Section 12. The function of the outprc subroutine is to put the data specified in the output sub-op into the buffer chain whose origin is in t.ocur. If no such chain exists, outprc must start one.

For any output control other than outmsg, the subroutine insert is called to add the specified characters to the output chain. If insert has to allocate a new buffer, it turns on the bffctl flag in that buffer; such a buffer is not included in t.ocnt.

The outmsg control causes the buffer chain whose head is pointed to by t.ocp to be appended to the chain at t.ocur. At this time tfwrit is turned on so that dia_man can tell that output is in progress, and t.ocnt is incremented according to the length of the new chain; if t.ocnt is not now over the buffer threshold, a request to the CS for more output is queued.

For most synchronous line types it is necessary to send complete messages to the channel, and to hold on to them until

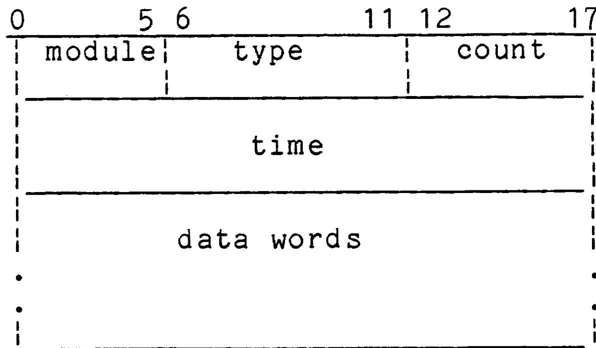
they have been acknowledged; therefore the t.ocur chain, if present, should never contain more or less than one complete message. This is enforced by use of the bffhld buffer flag, which is set by a holdot op block. When outprc sees bffhld on in the first buffer in the t.ocp chain, it copies t.ocp to t.ocur rather than threading into an existing chain. (It is an error if t.ocur is nonzero; this is why an outmsg control for a "held" buffer chain cannot be combined with any other output control, as explained in Section 12.) In this case, t.olst is set to point not necessarily to the last buffer in the t.ocp chain but to the first buffer in that chain whose bfflst ("last buffer") flag is on; t.ocnt is not incremented, no "send_output" request is queued, and tfwrit is not changed. This prevents dia_man from appending further output to the t.ocur chain; it is the responsibility of the control tables to issue send_output requests for complete messages as necessary.

TRACING

At various places in the FNP software, trace macros have been inserted to generate calls to the trace module. These calls result in entries being added to the memory trace buffer.

The memory trace buffer is a circular buffer allocated at the end of the trace module; its size is determined by the "size" statement associated with a "type: trace" statement in the FNP bindfile. (See the description of bind_fnp in Section 17.) The lowest address in the trace buffer is kept in .crtrb in the system communications region; .crtrc points to the oldest entry in the buffer, and is initialized to be equal to .crtrb. Once the trace buffer has been filled once, new entries overwrite the oldest ones, and .crtrc is advanced when such overwriting occurs. The entry at the highest address is followed by a word containing the "physical end" pattern, which is 525250 (octal); the entry most recently added to the buffer is followed by a word containing the "logical end" pattern, 525252. Thus, any program that interprets the trace buffer (as described in Section 16) starts at the address contained in .crtrc and processes entries until the physical end pattern is found; it then starts over at .crtrb and continues processing entries until it encounters the logical end pattern.

An entry in the trace buffer has the following format:



where:

module
: is the number of the module that made the trace call;

type
: is a number identifying the trace type within the module;

count
: is the number of data words associated with the entry (which may be zero);

time
: is the low-order 18 bits of the FNP clock at the time of the trace call;

data words
: are a variable number of words of optional associated data.

The module and type numbers are used in conjunction to determine the message to use when printing the trace buffer from a dump (see Section 16). In addition, trace checks the module number against the trace enable mask, which is specified in the bindfile and kept in .crtra in the system communications region; if the bit in .crtra that corresponds to the specified module number is not on, no entry is added to the trace buffer. The normal setting of the trace enable mask is either 357777, which allows tracing of every module except the scheduler and lsla_man, or 317777 which excepts the utilities module as well. The most common and useful method of examining the trace buffer is by means of the print_trace request to the debug_fnp command (see Appendix B).

SECTION 15

LOADING AND INITIALIZATION

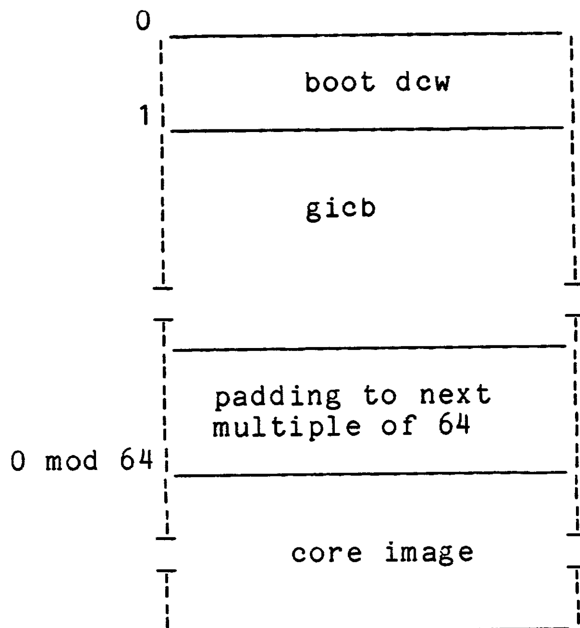
The loading and initialization of the FNP software is accomplished in four separate phases. First, the bind_fnp command combines the individual object modules and produces a simple core image of the FNP. The load_fnp_ subroutine, called by the answering service, patches the core image with configuration data obtained from the channel definition table (CDT) and starts the bootload of the FNP. The first program run by the FNP in response to the bootload interrupt is gicb, the intercomputer bootload module. This routine checks the DIA status and configuration and completes the loading of the FNP software if the configuration is correct. When the FNP software has been loaded gicb transfers control to init, the FNP software initialization routine. The init routine checks the configuration of the communications adapters, and prepares data bases for their use.

bind_fnp COMMAND

The bind_fnp command is similar to the Multics binder; it takes object modules and a bindfile and produces a bound segment. The object modules are produced by the FNP assembler, invoked by the map355 command. The bindfile consists of statements describing the maximum configuration of the FNP to be loaded, the modules to be included in this core image, and the size of certain tables. The output of bind_fnp is a segment containing an FNP core image. This core image has space allocated for the interrupt vectors, fault vectors, LSLA and HSLA hardware communications regions, IOM table and LSLA and HSLA tables. All of these tables are uninitialized with the exception of the IOM table which indicates the maximum configuration this core image can support. The bind_fnp command is designed to allow tailoring of the final core image to support the exact number and type of channels configured on the destination FNP. A command description of bind_fnp appears in the MAM Communications, Order No. CC75.

load_fnp_ SUBROUTINE

The load_fnp_ subroutine prepares the core image for bootload into the FNP. A segment is created in the process directory to contain the bootload program, gicb, and the core image. This segment is laid out as follows:



The copy of gicb from the system tape in >system_library_1 is copied into the boot segment in the process directory. The core image specified in the CDT is also copied into the boot segment.

The gicb routine contains a bootload communications area in its last 32 words which must be filled in before the FNP is bootloaded. This communications area contains the DIA list ICW and DCW list to be used to read in the core image, the mailbox address and terminate and emergency interrupt cell values which are checked against the configuration switches on the DIA, the load limits (high and low) for the core image, and checksums for gicb and the core image.

The load_fnp_ subroutine calls ring 0 to get the mailbox address and interrupt cell values and puts them in the bootload communications area. The core image load limits are copied out of the core image segment where the bind_fnp command left them, and stored in the bootload communications region.

Next, the CDT is scanned for each channel on the FNP being loaded, and information about the configuration of the channel is saved in an array. Also, the number of LSLAs and HSLAs required to support these channels is computed. Then load_fnp_ searches the IOM table in the core image and determines if the core image can support the required number of LSLAs and HSLAs. If not, the load attempt is aborted. If this test succeeds, the LSLA table is filled in by time slot with the desired configuration of the LSLA. When the FNP initialization routine runs, it will compare the actual configuration to the desired configuration and report any errors.

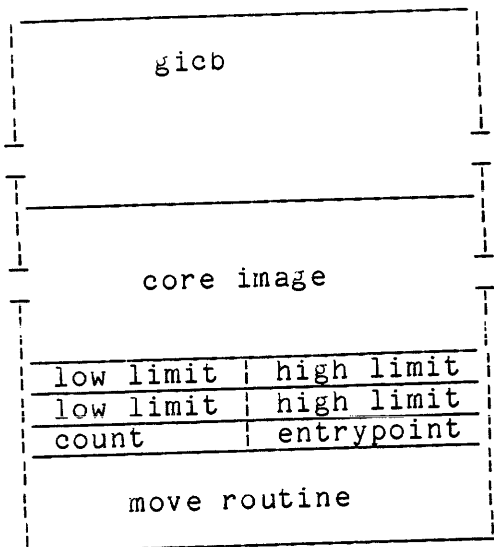
The information for HSLAs is processed in a similar fashion, with configuration information patched into the HSLA table. Also, for HSLAs a configuration PCW is set up for each channel based on the channel configuration. This PCW is stored in the HSLA hardware communications region for the channel where the FNP initialization routine will use it to configure the subchannel. For asynchronous channels, a basic PCW model is assigned which specifies two send ICWs, CCT enable, transmit even parity, and no receive parity check. This PCW is modified based on subchannel configuration and baud rate. If it is an EBCDIC code channel the character length is set to 7 bits (6 data plus 1 parity) and the parity generate and check is turned off. For non-EBCDIC channels the character length is set to 8 bits (7 data plus 1 parity). Finally, the baud rate of the channel is set in the PCW from the configuration information. For synchronous channels, the HSLA table is filled in as above, with information extracted from the CDT. The HSLA PCW is created by the internal procedure process_line_type. This procedure dispatches on the line type of the channel to set up configuration PCWs.

The communications region in the core image is modified to indicate the actual number of LSLAs and HSLAs, the date and time of bootloading, and the setting of the console enabled switch (from the check_switch argument to load_fnp_). Then the checksum of the core image is computed and saved in the bootload communications region of gicb. The segment is wired to get absolute addresses and the DIA list ICW and DCWs are filled in. The FNP addresses in the DCWs are assembled into gicb and need not be modified. The disconnect DCW is filled in and parity is computed on all DCWs in the list. The list ICW tally is set to the number of DCWs in the list. The checksum of gicb is computed and stored in the last word of the bootload communications area and the bootload is started by a call to hphcs_$load_fnp. This routine must fill in the bootload PCW in the mailbox and issue a connect to the DIA. At this point, load_fnp_ is done; it saves a pointer to the boot segment so that it may be deleted later, and returns.

gicb ROUTINE

The connect to the bootload PCW causes the DIA to read the gicb routine into the FNP under control of the boot ICW, which is the first word of the boot segment. The DIA then loads a DIA list ICW from the first location loaded by the bootload ICW (location zero) and proceeds as if a connect had been received from the FNP. The DCW list pointed to by this list ICW contains one DCW, a disconnect DCW. This disconnect causes a DIA status store and terminate interrupt. The terminate interrupt causes a tsy indirect through the interrupt vector at location 102(8), which has been loaded with the address of the entry into the bootload program. The FNP has now been started and gicb is running.

The gicb routine is entered at location 1000(8), where it tests the DIA status for errors and computes a checksum on the unmodified parts of itself, after correcting certain modifications. This checksum should be equal to the one computed by load_fnp_ and stored in the bootload communications area. The bootload communications area is then moved into the program from the end to its final location. Next, gicb reads the DIA configuration and compares it against the configuration supplied in the bootload communications area. If they disagree, the bootload is terminated. If the configuration is good and the DIA is properly set up, gicb connects to the DIA DCW list to read in the core image, starting at location 1000(8), and waits for the terminate interrupt. When it occurs, DIA status is checked and the checksum of the core image is computed and checked. If these are good, a small move routine is moved to the end of the core image (just following the program limits area) as follows:



Then gicb transfers to the move routine, which moves the core image to the location specified by the program limits area, clears the rest of memory and transfers to the entry point in the core image. This entry point is specified in the bind_fnp bindfile and is normally in the module init.

init MODULE

The init module is responsible for TIB and software communications region allocation for all configured channels, initialization of the HSLAs, initialization and startup of the LSLAs, and initialization of the DIA. It uses the IOM table to determine which IOM channels are supported by the core image being run, and the HSLA and LSLA tables as modified by load_fnp_ to determine how each communications channel is to be configured. During its operation, any errors encountered are reported on the FNP console if one is configured, and if the core image can support it (as determined by the value of .crcon in the system communications region).

First, the buffer pool is initialized to start at the end of init. The interrupt vectors are saved so that init can handle interrupts itself from the devices it is initializing without bringing the scheduler mechanism to bear, and a test is made on each IOM channel to see if the corresponding device is present. An "initialize" PCW is sent to each HSLA so that I/O can be done later on each subchannel. Then init cycles through the IOM table so that it can do further initialization for those devices that require it: the DIA, the LSLAs and the HSLAs.

DIA Initialization

The DIA initialization subroutine stores the address of the DIA "terminate" jump table in the interrupt vector save area, and the addresses of the 16 "special" interrupt jump tables located in dia_man in the interrupt vectors that are used for interrupts from the CS. (See Section 13 for details.) It then reads the DIA configuration switches, placing in a predetermined location in dia_man the address in CS memory of the FNP mailbox area and the interrupt levels on which the CS expects to receive FNP interrupts. Once it has done this, it sets up a DIA DCW list to send status to the CS informing it that init has been entered; the CS can then unwire the segment from which the core image was bootloaded (see above) immediately, rather than waiting until FNP initialization is complete.

HSLA Initialization

The HSLA initialization routine is called once for each HSLA specified in the IOM table. For each possible subchannel on the HSLA, it picks up the configuration PCW stored in the corresponding hardware communications region by load_fnp_; if this is zero, the subchannel is not configured. If the subchannel is to be configured, the address of the corresponding jump table in hsla_man is stored in the interrupt vector for that subchannel; an "unmask" PCW is sent to the subchannel, followed by the configuration PCW mentioned above. Then init sends a "request configuration status" PCW and waits 10 milliseconds for the status to be stored; this status is checked to see if it corresponds to the specified configuration. If it does, a software communications region and TIB for the subchannel are allocated and initialized (the initialization of a TIB is discussed later in this section). The modem type and flags specified in the HSLA table entry are used to determine if any flags have to be set in the TIB and/or the software communications region, and the line type, if specified, is stored in the TIB, overriding the default based on the baud rate (see "TIB Initialization" discussion below).

If a subchannel is not supposed to be configured (a zero configuration PCW has been supplied), the exhaust bit is set in the status ICW in the hardware communications region. If the configuration status obtained above does not reflect the intended configuration, or no configuration status was stored, a message is printed (if possible) on the FNP console, and a PCW is sent to mask the subchannel.

LSLA Initialization

The LSLA initialization routine is called once for each LSLA specified in the IOM table. It first sets the number of time slots associated with the LSLA according to the speed specified for the LSLA in the IOM table (normally 4800 bits per second, although 2400 is theoretically possible). It then ensures that the LSLA is running properly by putting it in receive mode and checking to see that incoming input frames begin with STX characters; the distance between successive STX characters is used to determine if the LSLA is running at the specified speed. A series of output frames is sent with a special configuration request sequence in the first (T & D) time slot; this results in an input frame being stored that describes the configuration of each time slot. (See Section 13 for an explanation of the relationship between time slots and communications channels.) The baud rate of each channel is determined by observing how many slots in succession contain the same configuration information. (The low-order bit of the slot alternates between on and off from one channel to the next, so if two adjacent slots are equal they must belong to the same channel.) The configuration information in each slot is compared with the information placed in the LSLA

table by load_fnp_; if they disagree, FNP initialization is aborted, unless the disagreement results from the complete absence of a channel either from the LSLA table (and hence the CDT) or from the configuration frame. A TIB is allocated and initialized for every channel thus validated.

Initial input frame buffers are allocated, and the ICWs in the hardware communications region are initialized to point to them. The initial input frame generally contains two SYN characters instead of the usual four; the initialization of the primary receive ICW reflects this. The tally for future input frames is taken from sf.ity in the software communications region, which is set to the normal frame length.

Send ICWs in the hardware communications region are set to point to the output buffers that are assembled into lsla_man following the software communications region. The address of the jump table for this LSLA is stored in the appropriate terminate interrupt vector; those fields in the software communications region that are not preset at assembly time are initialized. The starting address of the LSLA table is stored in the software communications region, and the output frame buffers are initialized with the LSLA "fill" characters.

TIB Initialization

When a TIB is allocated for an HSLA or LSLA channel, certain fields are filled in. The default line type of the channel is determined by searching the device table in the control_tables module (see Section 12) for the specified baud rate. The address of the first op block in the control tables to be executed is stored in t.cur, and certain flags are set according to the information in the device table entry for the line type of the channel. The address of the newly-allocated TIB is added to the end of the TIB table at the beginning of init.

Completion of Initialization

Once the initialization of all I/O channels is complete, init reports successful initialization to the CS (see below, "Status Reporting"). All of the init module from the end of the TIB table on is "freed" so as to be available for use as buffer space later. The interpreter is called at the "test-state" entry to start off the control tables for each configured channel. Each configured LSLA is put into send and receive modes. The interrupt vectors for the interval and elapsed timers are set up, interrupts are enabled, and init exits by transferring to the master dispatcher, which will wait until an interrupt arrives (see Section 11).

Status Reporting

During FNP initialization, status is reported to the CS over the DIA by storing a 36-bit status in word 6 of the mailbox header in CS memory and sending an interrupt to the CS. This status has the following format:

bit	meaning
0	on if valid FNP bootload status
1-2	not used
3-5	major status
6-8	not used
9-17	minor status (when error reported by init)
18-35	IOM channel (when error reported by init)

The major status has one of the following values:

- 0 initialization completed successfully
- 1 checksum error in core image
- 2 I/O error reading core image
- 3 error reported by gicb
- 4 error reported by init
- 5 init entered (wired segment can be released)

If the major status is 4, the minor status is used to find a more detailed error message in dn355_messages, and the IOM channel indicates which channel on the FNP IOM was being initialized when the error was detected.

SECTION 16

FNP CRASH ANALYSIS

HOW THE FNP CRASHES

The FNP ceases operation as a result of most faults, as described in Section 14. Some of these faults are hardware-induced, such as memory parity, IOM channel faults, illegal interrupts, etc.; others are more likely to be caused by faulty software. The most frequent cause of FNP crashes is an inconsistent or unacceptable condition detected by the Multics Communication System software, which then deliberately executes an illegal operation code contained in a specially coded word. This word, generated by a "die" macro, contains a number identifying the module containing it and a code identifying the particular condition detected. The fault code, the FNP instruction counter, and the faulting instruction itself are all stored in words 6 and 7 of the mailbox header in CS memory, and an "emergency" interrupt is sent over the DIA. The information in the mailbox header is used by dn355 to write a message on the syserr console describing the crash; dn355 also sends a wakeup to the process that bootloaded the FNP (generally the initializer) to inform it of the crash.

DUMPING THE FNP

fdump_fnp_

When the initializer is told that the FNP has crashed, it calls fdump_fnp_ in order to dump the contents of FNP memory into CS memory. Such a dump can also be initiated manually be means of the operator command fdump_fnp. In either case, fdump_fnp_ creates a segment in >dumps with an entryname derived from the FNP tag and the date and time of the crash; it then calls hphcs_$fdump_fnp (a gate entry leading to fnp_util$fdump) to do the actual dumping. This ring 0 procedure uses fnp_dump_seg as a buffer for the contents of the dump.

Dumping the FNP is one of the two cases in which I/O over the DIA is initiated by the CS (the other one is bootloading the FNP). The CS address provided in the DIA PCW points to a control word containing the FNP address and tally to be used to dump data into CS memory immediately following the control word. Since this control word is the third word of fnp_dump_seg, FNP memory is read in pieces of 1021 words; after each interrupt from the DIA, a 1021-word piece of data is copied into the segment created by fdump_fnp_, until all of FNP memory has been read.

FD355 and DMP355

If Multics crashes and it is desirable to examine the contents of FNP memory, a dump can be taken by means of either of the BOS commands FD355 or DMP355. FD355 creates a dump (in the dump partition) that is later copied into >dumps with a name derived from the current ERF (error report form) number like that of an FDUMP, but with a final component of "355". DMP355 dumps the contents of the FNP directly to a printer. See the System Dump Analysis PLM, Order No. AN53, for more information.

Obtaining A Printed Dump

An FNP dump in >dumps can be converted to a form suitable for printing by use of either the online_dump_fnp command or the online_dump_355 command. By specifying "file_" for the -dim control argument and a pathname for the -device control argument, the user produces a formatted copy of the dump that can then be dprinted. The online_dump_355 command can only be used to format dumps created by FD355 (see above); online_dump_fnp may be used to format dumps produced by either of the methods described above, but if it is used on a dump created by FD355, the -pathname control argument is necessary to specify the name of the dump. See the descriptions of the two commands in Appendix B.

In general, a printed dump is not needed; the debug_fnp command (described in Appendix B) can be used to analyze the dump online. A printed dump might be useful if the person analyzing the dump is unable to log in to the site at which the dump was taken, or if it seems necessary to scan the contents of FNP memory because it has been completely or largely overwritten.

INTERPRETING AN FNP DUMP

The interpretation of an FNP dump is normally carried out by means of the debug_fnp command. The paragraphs below describe the format of a printed dump in case that is all the analyzer of the dump has available. References are nonetheless included to the debug_fnp requests that may be used to obtain the information described.

Format of the Dump

The header of a printed FNP dump identifies the segment containing the dump and the FNP that was dumped; this is followed by the date and time that the core image was created by bind_fnp and the date and time that the FNP was last bootloaded. If the FNP crashed as a result of a "die" macro (as described above), the "crash reason" derived from the contents of the "die" word is printed. Then comes a line identifying the fault that precipitated the crash ("illegal opcode" in the case of a software-induced crash) followed by the contents of all the machine registers at the time of the fault.

The next item is the module chain, a list giving the starting address of each FNP module identified by its "short" name (see Section 9 for a discussion of module names). This is followed by the "trace table," containing messages describing all the events appearing in the circular trace buffer, oldest first (see the description of the memory tracing mechanism in Section 14).

The remainder of the dump presents the entire contents of FNP memory, eight 18-bit words per line. Each line contains the following:

- absolute address of the first word on the line (in octal)
- the name of the module containing the line (this field is blank for low memory before the start of the first module)
- relative address within the module (in octal)
- the octal contents of the eight words starting at the specified address
- the ASCII representation of the same eight words (characters that cannot be represented are replaced by blanks)

Duplicate lines are not printed; a star (*) following the absolute address indicates that duplicate lines immediately preceding the current line have been omitted. Appendix D contains a detailed description of the layout of FNP memory.

Crash Reason

The "crash reason" message, identical to the message printed on the syserr console when the FNP crashes, indicates what condition the FNP software detected. It is printed in response to the why request to debug_fnp. If it is any of the messages listed below, an FNP hardware problem is indicated:

dia_man: unrecoverable I/O error

dia_man: more then 5 consecutive I/O errors

dia_man: 3 consecutive mailbox checksum errors

hsla_man: receive transfer timing error

hsla_man: xmit transfer timing error

lsla_man: send transfer timing error

lsla_man: more than 10 successive re-sync attempts

Other messages, indicating software errors, are apt to be self-explanatory, but a few require some clarification.

A message saying "buffer allocation failed" generally indicates that some channel or set of channels has gone out of control allocating buffers, and the mechanisms intended to prevent this have failed. The crash in this case almost always occurs in lsla_man (if any LSLAs are configured), since an LSLA input buffer must be allocated every 100 milliseconds for each LSLA, as described in Section 13; it does not necessarily indicate a failure of the LSLA software. It may be, in fact, that an attempt is being made to run more channels than a single FNP can handle, particularly if a large number of HSLA channels are configured; HSLA channels are particularly expensive in terms of buffer space.

Crash messages from the interpreter indicate probable errors in a control tables module, particularly if installation-supplied or installation-modified control tables are being used. It may be helpful in such cases to note that index register 2 usually contains the address of the current op block when the interpreter is running, and that index register 1 contains the current TIB address. The interpreter message "type not of form 777xxx" indicates an attempt to execute an op block that is not really an op block; in particular, if index register 2 contains 776(8), an attempt was made to transfer into a control tables module not included in the core image. The last few entries in the trace table (see below) are likely to be useful in determining how this happened.

Various error messages produced by the utilities module indicate errors detected by the buffer-freeing routine. In some of these cases it is useful to check the free space chain starting at .crnxa, described in Section 14. Most often, however, it is more interesting to find out what routine called frebuf, and what it was trying to free; see "Tracing Subroutine Calls," below, for details. Note that if the error message is "tried to free buffer with address < .crbuf", the offending address is in index register 3.

Fault Identification

If the fault name that appears in the dump is "illegal opcode," "overflow," "store fault," or "divide check," a probable software problem is indicated. If an illegal opcode was generated by a "die" macro, a crash reason message also appears, as described above. In all other cases of software failure, it is probably necessary to go through the instruction counter and find out what was being executed at the time of the fault.

If the crash was caused by an operator typing "ABORT" on the FNP console (see Section 13), the fault identification line in the dump simply says "abort." If the fault is anything other than those mentioned so far, a hardware problem is indicated.

If the FNP did not crash, but was dumped either by an operator fdump_fnp command or by the FD355 command after a Multics crash, the fault identification is "none."

Machine Registers

The line after the fault identification gives the octal contents of the machine registers at the time of the fault, in the following order:

instruction counter

indicator register

A register

Q register

index register 1

index register 2

index register 3

interrupt enable register

elapsed timer register

These values are saved in an array of locations in the utilities module by the fault-handling software, as described in Section 14; accordingly, if there was no fault (fault indicator is "none"), they appear in the dump as all zero. They are printed in response to the regs request to debug_fnp.

Some common uses of index registers are described here, since they may be useful in pinpointing problems. When any work that is specific to a channel is being done, the address of that channel's TIB is virtually always in index register 1. Index register 2 is generally used by the interpreter to point to the current op block; hsla_man normally keeps the address of the software communications region in index register 2. The address of a buffer or buffer chain being allocated or freed is passed to the buffer freeing routine or returned by the buffer allocation routine in index register 3.

Trace Table

The trace table reports the contents of the memory trace buffer at the time of the fault, and thus describes the events immediately preceding the crash. The types of events normally appearing in the trace include: all DIA transactions; interrupts from, and status reported by, all active HSLA subchannels; and all calls to the interpreter and the op blocks executed as a result of such calls. If the failure seems to be in connection with a particular channel, it is often useful to examine only those trace entries that reflect events related to that channel, thus effectively obtaining a history of the channel for up to several seconds before the crash. All or part of the trace table can be printed by means of the print_trace request to debug_fnp.

Tracing Subroutine Calls

Having determined by means of the instruction counter what subroutine the FNP was executing in at the time of the fault, it is sometimes useful to find out how it got there. In most cases, the first word of the currently executing subroutine (whose address can be found with the help of program listings) contains the absolute address of the location to which the subroutine was expected to return. Once this location is found in the dump, the module name and relative address can be used to determine what routine made the call; if necessary, the first word of this

routine can be examined to find out where it was called from, etc. This can be done automatically using the call_trace request to debug_fnp.

Other Useful Information

See Section 10 for a description of various FNP data bases, Appendix B for a description of the debug_fnp command, and Appendix D for a description of the layout of FNP memory.

SECTION 17

FNP-RELATED COMMANDS

This section describes the operations of various Multics commands that deal with FNP core images and dumps. The usage of all these commands is described in Appendix B, except for map355, which is described in the MAM Communications, Order No. CC75. The commands described in this section are divided into two groups: those that are used in core image preparation, and those that are used in analyzing FNP dumps.

CORE IMAGE PREPARATION

The commands described here are map355, which is used to invoke the 355MAP assembler, and coreload, which tranforms a single FNP object segment into an FNP core image. An additional command, bind_fnp, binds a collection of FNP object segments into a core image; its operation is described in Section 15.

map355

The map355 command prepares a GCOS job deck for processing by the GCOS environment simulator, which it then invokes by calling the gcos command. The job deck is created in a segment in the process directory with an entryname of NAME.jobdk_, where NAME is the name of the module to be assembled, truncated to 11 characters (if it is longer) to allow for additional suffixes. For more information on GCOS job decks, see the Multics GCOS Environment Simulator manual, Order No. AN05.

The GCOS simulator, when invoked by map355, runs the 355MAP assembler, using a source segment with the entryname NAME.map355, where NAME is the name of the module (not truncated), and produces an object segment having the form of a GCOS binary card deck. Such a deck can be read by the GCOS utility gcos_gsr_read_ and interpreted by commands such as bind_fnp and coreload. This segment is created in the working directory with an entryname of NAME.objdk. In addition, a GCOS listing file (in BCD) is produced in either the working directory or the process directory, depending on whether or not the -gcos_list control

argument is specified; its entryname is NAME.glist in the working directory or NAME.glist_ in the process directory. In either case, map355 converts this file to an ASCII listing file by calling gcos_sysprint. The result of the conversion is placed in the working directory with an entryname of NAME.list if the -list control argument is specified; otherwise it is placed in the process directory with the entryname NAME.list_.

After converting the listing file, map355 searches it for a line of the form:

there were N warning flags in the above assembly

where N is either the word "no" or the number of errors detected by the 355MAP assembler. This line is printed on the user's terminal. If any errors were detected, map355 searches the file for lines containing warning flags (identified by a letter in the first column) and prints each such line found on the terminal. Finally, it deletes all temporary files created in the process directory by either map355 or the GCOS simulator and two temporary files created by the simulator in the working directory.

<u>coreload</u>

The coreload command uses an object deck produced by the map355 command and produces a segment suitable for loading into and execution by an FNP. The input object deck has an entryname of NAME.objdk, and the output core image segment has an entryname of NAME. NAME.objdk must be an absolute object deck, i.e., it must contain no relocatable text. This can be ensured by the presence of an abs pseudo-operation in the source segment (NAME.map355).

The coreload command reads the object deck one card image at a time by calling gcos_gsr_read_. Each card is identified as either an absolute text card or an end-of-deck card. (Any other type of card is reported as an error.) Each text card contains one or more blocks of text, where each block is preceded by a header containing the starting address of the block and the number of 18-bit words of text in the block. The header information is used to copy the text from the card image to the corresponding address in the output segment, and to find the next block on the card, if any.

When either an end-of-deck card is encountered or the input segment is exhausted, coreload calculates the number of 36-bit words in the core image by taking half of the last 18-bit address containing text; this count is stored in the first 36-bit word of the output segment, and is used to set the bit count of the segment.

DUMP ANALYSIS

The online_dump_fnp and online_dump_355 commands are used to produce an ASCII representation of an FNP dump that was generated as described in Section 16. Both commands call online_355_dump_ to process the dump; the difference between them is that online_dump_fnp can process dumps created by the fdump_fnp initializer command with names of the form fnp.TAG.DATE.TIME, whereas online_dump_355, which is an entry in the online_dump command, only recognizes dumps generated by the BOS command FD355 with names of the form DATE.TIME.N.ERFNO.355. The output of online_355_dump_ is written to a stream using ios_; the attachment of the stream is determined according to the -dim and -device arguments specified in the command line. Output is usually directed through the file_ DIM to a file that can then be printed or examined online using an editor. For more information on interpreting the dump, see Section 16.

The debug_fnp command is used for online FNP dump analysis, as well as for displaying and interpreting the contents of either the memory of a running FNP or a core image in the Multics virtual memory. It can be used on dumps created by either fdump_fnp or FD355. It uses a database derived from the source of the Multics Communication System macro library (macros.map355) in order to recognize the symbolic names of fields in the TIB, software communications region, etc., and to print comments in response to the explain command. Some specially-coded comment lines are included in the macro source in order to identify the information required by debug_fnp. For details on the usage of debug_fnp, see Appendix B.

APPENDIX A

MAILBOX OPERATION CODES

This appendix lists all the operation codes placed in the submailboxes used for communication between the CS and the FNP. For each operation code, the following information is given: the octal value of the code, its purpose, and a description of the associated data, if any, passed elsewhere in the submailbox.

OPERATION CODES SENT FROM THE CS TO THE FNP

Operations Sent with a WCD I/O Command

Terminal Accepted (000)

Purpose: Acknowledge the connection of a channel.

Associated Data: Word 2: Bits 0...17 contain the output buffer threshold; FNP sends a "send output" operation when output chain falls below this size.

Disconnect Line (001)

Purpose: Instruct the FNP to hang up the channel.

Associated Data: None

Disconnect All Lines (002)

Purpose: Hang up any currently connected channels, stop accepting dialups. Data terminal ready is turned off for all channels; no further DIA I/O is done until the FNP is reloaded.

Associated Data: None

Don't Accept Calls (003)

Purpose: Instruct the FNP to ignore dialups until the next accept calls operation.

Associated Data: None

Accept Calls (004)

Purpose: Start allowing dialups.

Associated Data: Word 2: Bits 0...17 contain absolute address of the end of the circular buffer in tty_buf.

Set Line Type (006)

Purpose: Change the line type of a channel.

Associated Data: Word 2: Bits 0...17 contain the new line type.

Enter Receive Mode (007)

Purpose: Cause a channel using the TermiNet 1200 interface with a Bell 202C modem to start receiving input data.

Associated Data: None

Set Framing Characters (010)

Purpose: Provide the frame_begin and frame_end characters to be recognized when in blk_xfer mode.

Associated Data: Word 2: bits 0...8 contain the frame_begin character.

bits 9...17 contain the frame_end character.

Blast (011)

Purpose: Send a specific message to all connected terminals (used by the BOS BLAST command).

Associated Data: Word 5: bits 0...17
contain the absolute address of a block of 96 36-bit words that consist of 32-word output buffers containing the message in ASCII, EBCDIC, and correspondence codes.

Dial Out (014)

Purpose: Dial out over a channel attached to an automatic call unit.

Associated Data: Word 1: bits 0...17
contain the number of digits in the phone number.

Words 2-3
contain the digits of the phone number, 6 bits for each digit.

Reject Request (016)

Purpose: Report that there is insufficient room in the circular buffer for the channel's input (see "Accept Input"). The FNP will retry the input request after one second.

Associated Data: None

Terminal Rejected (020)

Purpose: Refuse a connection requested by the FNP by means of an accept new terminal operation (see below), either because of lack of space in tty_buf or because the channel is not in the "listening" or "dialing-out" state.

Associated Data: None

Disconnect Accepted (021)

Purpose: Acknowledge a line disconnected operation (see below) sent by the FNP.

Associated Data: None

Dump Memory (023)

Purpose: Copy a specified portion of FNP memory to Multics memory.

Associated Data: Word 2
contains the absolute address of the CS buffer to which the data is to be copied.

Word 3: Bits 0...17
contain the starting FNP address from which data is to be copied.

Bits 18...35
contain the number of 18-bit words to be copied.

Patch Memory (024)

Purpose: Replace the contents of a specified portion of FNP memory.

Associated Data: Word 2:
contains the absolute address of the buffer containing the data to be patched into the FNP.

Word 3:
contains the FNP address and tally as described for Dump Memory (above).

Set Break (025)

Purpose: Set, reset or restart a breakpoint in a control tables module.

Associated Data: Word 2: bits 0...17
contain the FNP line number of the channel to which the breakpoint is to apply, or all ones if the breakpoint is to apply to all channels.

bits 18...35
contain the FNP address of the breakpoint.

Word 3: bits 0...17
contains 1 to set a breakpoint; 2 to reset a breakpoint; or 3 to restart execution suspended at a breakpoint.

bit 18
is "1"b if memory tracing is to stop when a channel hits the breakpoint.

Line Control (026)

Purpose: Send line control information to a control tables module.

Associated Data: Words 2-4
contain the 72 bits of line control information.

Synchronous Message Size (027)

Purpose: Inform the FNP that the input messages from a synchronous channel are expected to be no larger than a specified size. This enables the FNP to allocate buffers of an appropriate size.

Associated Data: Word 2: bits 0...17
contain the message size in characters.

Break Acknowledged (035)

Purpose: Acknowledge a line break operation (see below) sent by the FNP.

Associated Data: None

Alter Parameters (042)

Purpose: This operation code is used as a mechanism for extending the set of operation codes, particularly in connection with mode changes. The precise function of the operation depends on the subtype; the various Alter Parameters subtypes are described later in this appendix.

Associated Data: Word 2: Bits 0...8
contain the subtype. Other data depends on the subtype.

Checksum Error (043)

Purpose: Report that the submailbox most recently sent by the FNP contained an incorrect checksum.

Associated Data: None

Set Delay Table (045)

Purpose: Provide a delay table to be used by the channel when echoing carriage movement characters.

Associated Data: Words 2-4
contain the six delay values, 18 bits for each value.

Operations Sent with a WTX I/O Command

Accept Output (012)

Purpose: Inform the FNP that output is available for a specified channel.

Associated Data: Word 5: Bits 0...17
contain the absolute address of a list of "pseudo-DCWs" giving the addresses and tallies to be used in reading the output data.

Bits 18...35
contain the number of pseudo-DCWs in the list.

Accept Last Output (013)

Exactly like accept output (above) except that its use indicates that the output being sent is at the end of a 6180 write chain.

Operations Sent with an RTX I/O Command

Input Accepted (005)

Purpose: Respond to an accept input operation (see below) by providing the address (in the circular buffer) to which input is to be sent.

Associated Data: Word 5: Bits 0...17
contain the beginning absolute address of the portion of the circular buffer into which the input is to be placed.

Bits 18...35
contain the number of characters to be placed in the specified location.

Word 4:
If nonzero, contains the address and tally as described above for the remaining data. This word is only used if the input request

required a wraparound of the circular buffer.

OPERATION CODES SENT FROM THE FNP TO THE CS

Operations Sent with an RCD I/O Command

Accept New Terminal (100)

Purpose: Report that a dialup has been received on a channel.

Associated Data: Word 2
contains the line type of the channel.

Word 3
contains the baud rate of an autobaud channel, or 0.

Line Disconnected (101)

Purpose: Report that a channel has hung up.

Associated Data: None

Input in Mailbox (102)

Purpose: Transfer a short input message from the FNP to the CS.

Associated Data: Word 0: Bits 18...35
contain the number of 32-word blocks currently available in the FNP.

Word 1: Bits 9...17
contain the number of characters in the input message.

Words 2-26
contain the input message (up to 100 characters).

Word 27: Bit 16
is "1"b if an output chain is present in the FNP.

Bit 17
is "1"b if the input contains a break character.

Send Output (105)

Purpose: Inform the CS that the FNP is prepared to accept output for a channel.

Associated Data: Word 0: Bits 18...35
contain the number of 32-word blocks currently available in the FNP.

Accept Input (112)

Purpose: Inform the CS that input from a channel is available.

Associated Data: Word 0: Bits 18...35
contain the number of 32-word blocks currently available in the FNP.

Word 2: Bits 0...17
contain the number of characters of input available.

Bit 34
is "1"b if an output chain is present in the FNP.

Bit 35
is "1"b if the input contains a break character.

Line Break (113)

Purpose: Report that a line break condition has been detected on a channel.

Associated Data: None

"Wru" Timeout (114)

Purpose: Report that a channel did not respond to a "wru" operation (requested by an alter parameters operation with a wru subtype).

Associated Data: None

Error Message (115)

Purpose: Report an error condition and cause a message to be printed on the syserr console.

Associated Data: Word 2: Bits 0...17
contain a code indicating the type of error.

Bits 18...35
contain the first of up to three pieces of data to be used in constructing the error message.

Word 3: Bits 0...17
contain the second piece of data; bits 18...35 contain the third piece.

NOTE: The following 4 operation codes are used to report failure of a dialout attempt made as a result of a dial out operation (see above). None of them has any associated data.

No Power to ACU (120)
Data Line Occupied (121)
Dial Out Failed (122)
Unable to Dial Out (123)

Line Status (124)

Purpose: Report line status generated by a linsta op block.

Associated Data: Words 2-3
contain the 72-bit line status.

SUBTYPES USED WITH ALTER PARAMETERS OPERATIONS

In all cases, word 2, bits 0...8 of the submailbox contain the subtype.

Mode Changes

The following subtypes are all used to turn a specified mode on or off for a specified channel. Word 2, bit 17 is "1"b if the mode is to be turned on, or "0"b if it is to be turned off. Unless otherwise specified, these subtypes do not supply any other data.

Full Duplex (003)

Crecho (010) - carriage return echo

Lfecho (011) - linefeed echo

Lock (012) - turn keyboard and printer addressing on or off.

Tab Echo (016)

Listen (020)

Additional Data: if the mode is being turned on, word 2, bits 18...35 contain the size, in characters, of input buffers to be allocated for the channel.

Handle Quit (021)

Echoplex (024)

Transmit-hold (025)

Replay (027)

Polite (030)

Block Transfer (031)

Additional Data: Word 2: Bits 18...35 contain the size, in characters, of each input buffer to be used when not within a frame.

Word 3: Bits 0...17 contain the buffer size to be used within a frame.

Breakall (033)

Prefixnl (034)

Other Subtypes

Dump Output (015)

Purpose: Discard any untransmitted output.

Associated Data: None

Change Control String (022)

Purpose: Change index of strings used for keyboard and printer addressing.

Associated Data: Word 2: Bits 9...17 contain the index of the new set of control strings.

Wru (023)

Purpose: Send a "Wru" command to the channel in order to read a terminal's answerback.

Associated Data: None

Dump Input (026)

Purpose: Discard any pending input.

Associated Data: None

Set Buffer Size (028)

Purpose: Inform the FNP what size input buffer to allocate for the channel (used for dialout channels).

Associated Data: Word 2: Bit 17
is "1"b.

Bits 18...35
contain the size, in characters, of input buffers to be allocated for the channel.

APPENDIX B

COMMAND DESCRIPTIONS

This appendix consists of command descriptions for the following commands: coreload, debug_fnp, online_dump_fnp, online_dump_355, and tty_analyze. This appendix describes the use of these commands in the manner of the MPM. The operation of coreload, online_dump_fnp, and online_dump_355 is described in Section 17; that of tty_anaylze is described in Section 8. Other commands that were described in previous editions of this PLM are now described in the *MAM-Communications*, Order No. CC75: bind_fnp, map355, tty_dump, and tty_meters.

Name: coreload

The coreload command converts a single object program produced by map355 to a core image segment suitable for loading into an FNP. The object program must be absolute, i.e., it must not contain relocatable text.

Usage

```
coreload path
```

where path is the pathname of the object segment; if the suffix ".objdk" is not present, it is assumed. The result of the conversion is a segment in the working directory whose entry name is the same as that of the input object segment with the suffix ".objdk" removed.

Name: debug_fnp, db_fnp

The debug_fnp command is a debugging aid intended to be used by FNP software developers and in FNP dump analysis. The command can be used to patch or dump memory in a running FNP, to examine a dump from a crashed FNP or a core image segment before it is loaded, to set breakpoints in a running FNP, symbolically display FNP control blocks, buffers, etc.

Usage

debug_fnp {initial_request_line}

where initial_request_line specifies the first request(s) debug_fnp is to execute. If initial_request_line contains blanks or semicolons, it must be enclosed in quotes. Once the initial request(s), if any, are completed, debug_fnp reads request lines from user_input. Each line may contain multiple requests, separated by semicolons. If an error occurs in any request, the remainder of the requests on that line will not be executed. Any debug_fnp request can be aborted by issuing a "Quit" followed by a Multics "program_interrupt" command.

Selecting debug_fnp Mode

The debug_fnp command can be set up to operate on either a running FNP, a dump segment, or a core image segment. When first invoked, the command is set up to examine the first configured FNP. It is possible to switch betweens dumps, core images, and running FNPs at any time. With few exceptions, most debug_fnp requests work the same regardless of whether a running FNP, a dump, or a core image is selected.

To select a running FNP:

fnp tag

where "tag" is "a", "b", "c", or "d".

To select a core image:

image path

To select a dump:

dump path

where path is the Multics pathname of a segment containing the dump or the core image. Core image segments and dump segments have different formats, so these requests are not interchangeable. The pathnames on the dump and image requests can also be starnames, providing they match one and only one entry in the directory specified.

In most cases, it is not necessary to know the pathname of the dump to be examined, as special requests are provided for selecting dumps.

To list all the dumps currently in the dump directory:

dumps

The default dump directory is ">dumps" but this can be changed by:

dump_dir {path}

where path is the pathname of the new dump directory. If "path" is omitted, the name of the current dump directory will be printed.

To select the latest dump:

last_dump

The next earliest dump can be selected with:

prev_dump

The prev_dump request can be used repeatedly as long as there are more dumps.

To select the next latest dump:

next_dump

The next_dump and prev_dump requests can be used to peruse any or all of the dumps in the dump directory, going in either direction.

If dealing with a dump which contains multiple FNPs, such as a BOS fdump, the following request is used to select which FNP in the dump is examined:

select_fnp tag

where tag is "a", "b", "c", or "d".

To find out what FNP, dump, or core image is selected:

what

will print the FNP tag, or the pathname.

Expressions

Many of the following requests take numeric arguments such as addresses, lengths, etc. Any of these arguments can be expressed as a generalized FNP expression. Expressions can be arbitrarily complex, containing "(", ")", "+", "-", "*", and "/" with their normal meanings and precedence. The symbol "¦" is synonymous with "+", as in module¦offset. Indirection can be specified by ",*", following the address to indirect through. Numeric constants are interpreted as octal, unless they are followed by a ".", in which case they are decimal. The symbol "*" can be used for the current location counter, which is generally the last address used in a display or patch request. Many common FNP symbols can also be used, including all fields in the system communications region, the hardware communications region, the software communications region, and the TIB. (Note: before TIB, hwcm, and sfcm addresses can be used, the addresses of these control blocks must be established. See the "line" and "set" requests). A symbol may also be any op block mnemonic, the name of any FNP object module, or a machine instruction (specified by surrounding the instruction by apostrophes). In addition, user symbols can be defined. Examples of expressions:

hsla¦500	(offset 500(8) in hsla_man)
t.icp,*	(the contents of t.icp in the current TIB)
*+30	(30(8) words beyond the current location counter)
tib¦14,*+10	(10(8) words beyond the address contained in word 14(8) of the current TIB)

```
goto              (a goto op block code, i.e., 777001)
'lda 0,2,b.0'     (instruction word)
cax3              (apostrophe not needed if no operand)
```

DISPLAYING FNP MEMORY

To display the contents of FNP words:

```
display address {length} {mode}
d address {length} {mode}
```

where "address" is the starting address, "length" is the number of words, and "mode" is the display mode. The symbol "*" will be set to the address sepcified. The following display modes can be used:

```
octal, oct
character, ch
address, addr          (in form module|offset)
clock, ck              (4 FNP words as a Multics clock)
instruction, inst      (355 instruction format)
opblock, op            (pseudo opblock format)
decimal, dec
bit
ebcdic, ebc
```

If omitted, the length defaults to 1 unless "address" is a predefined FNP symbol, in which case the appropriate length for that symbol will be used. Similarly, if the mode is omitted, octal is used, unless "address" is a predefined FNP symbol in which case the mode appropriate for that symbol is used.

To display a buffer:

```
buffer {address} {mode} {-brief|-bf}
buf {address} {mode} {-brief|-bf}
```

where "address" is the address of the buffer, "mode" is the mode to display it in (see display request), and -brief means display only the first 2 words of the buffer. If "address" is omitted, the next buffer pointer from the previous buffer displayed is used. If "mode" is omitted, character mode is assumed. If -brief is not specified, the entire buffer is displayed. The length is determined automatically by reading the buffer header.

To display a buffer chain:

```
buffer_chain {address} {mode} {-brief|-bf}
bufc {address} {mode} {-brief|-bf}
```

where the arguments are the same as in the buffer request. This request will follow the threads in the buffer chain, displaying each buffer.

If the data being displayed is in the form of threaded control blocks, the following requests can be used:

```
block {address} {-offset|-o offset} {-length|-l length}
blk {address} {-offset|-o offset} {-length|-l length}
```

will display a control block at the address specified. The length of the block is specified with -length. The default is 8 words. The offset to the forward pointer in the block is specified with -offset. The default is 0. If the address is not specified, the next block in the chain will be displayed (using the forward pointer from the previous block).

To display an entire chain of control blocks:

```
block_chain {address} {-offset|-o offset}
  {-length|-l length}
blkc {address} {-offset|-o offset} {-length|-l length}
```

will display control blocks until one with a zero forward pointer is encountered.

If the data being displayed is a word of flags, the flags request can be used to show the setting of individual bits.

```
flags address {type}
```

where address is the the address of the word containing flags, and the type can be:

t.stat	TIB status word
t.flg	first TIB flag word
t.flg2	second TIB flag word
sf.flg	HSLA sfcm flags
istat	interpreter status word
hs.1	first word of HSLA hardware status
hs.2	second word of HSLA hardware status

If {type} is omitted, it is assumed to be the same as "address", which then must be one of the items in the above list. The flags are listed by name, as they appear in the macros.map355 source file. The explain request (see other requests) can be used to help with unfamiliar names. Occasionally, the value of a flag word is known (from a trace, for example), without knowing an address of it. In this case, the following form can be used:

```
flags =expression type
```

where expression is any valid expression, and type is one of the types shown above.

PATCHING FNP MEMORY

To patch the contents of FNP memory:

```
patch address arg1 ... {argn}
```

where address is the starting address to patch, and the arg_i represent patch data. Each arg_i may be an expression representing the value to be stored in 1 FNP word, or a character string in quotes (which may contain more that 1 word of data). The total number of words patched cannot exceed 32. Before the patch is applied, the effects of the patch are displayed (old and new contents of every word) and the user is asked to verify that the patch is correct. The symbol "*" will be set to the address specified. Examples of patch requests:

```
patch 43102 203456 -1 2
patch .crver "3.1x"
patch ctrl¦1400 goto ctrl¦1600
patch hsla¦1541 'tze 13' cax3 'lda 0,2,b.1'
```

A shorthand form of this request is:

```
=arg1 ... {argn}
```

which is equivalent to:

```
patch * arg1 ... {argn}
```

Individual flag bits in words of flags can be manipulated with the following requests:

```
set_flag flag_symbol
```

will set the bit associated with the flag_symbol specified in the appropriate word. In a similiar way,

    clear_flag flag_symbol

will clear an individual bit. Currently, these requests are not indivisible operations: this means that if other flags bits in the word are dynamically changing, these requests may change their value if they happen to have been changed between the time the word was read and when it was rewritten.

DUMP ANALYSIS REQUESTS

The following requests are only valid when using debug_fnp on a dump.

To find out the cause of a dump:

    why

will print the type of fault which caused the crash, and if the crash was caused by a "die" opcode in the FNP, will interpret the reason for the crash.

The request:

    regs

will print the contents of all machine registers at the time of the fault.

If the fault occured in a subroutine (as defined by the map355 "subr" macro), information about the call is available with:

    call_trace address {-long¦-lg}

This request will start at the address specified and perform a backward trace of all subroutine calls. If -long is specified, the registers saved at each subroutine level will also be printed. This request can also be used on a running FNP, but the information is probably changing too fast for the request to be useful.

FNP TRACE TABLES

A running FNP or a dump contains a trace table of the most recent events occuring in the FNP. The trace table can be displayed with:

```
print_trace {start}
print_trace {start} {count}
```

where start indicates the starting trace message and count is the number of messages to display. If no arguments are given, the entire trace table is printed. If no count is given, the trace table is displayed from the starting point specified to the end. If the start number is positive it is counted from the oldest message; if negative, it is counted from the most recent. For example:

```
print_trace 200.
```

will skip the 199 oldest entries and print the rest.

```
print_trace -50.
```

will print the 50 most recent messages.

Printing the trace table of a running FNP is only meaningful if tracing has been suspended; otherwise the table is changing too fast to be interpreted. Tracing can be suspended in a running FNP by:

```
stop_trace
```

and restarted with:

```
start_trace
```

Tracing can also be stopped and started with some of the breakpoint requests explained below.

Which modules in the FNP are traced is determined by the trace mask, kept in FNP memory. This mask may be examined or updated with:

```
trace_mask {modules}
```

If used with no arguments, trace_mask will display and interpret the current trace mask. If modules are given, they represent modules to be added to or deleted from the current mask. The

module should be specified as 'name' or '+name' to set the tracing bit for the module; it should be '-name' or '^name' to turn off the corresponding bit. In addition, all and none may be specified. For example:

```
trace_mask hsla ^dia -lsla
```

turns on tracing for hsla_man and turn off tracing for dia_man and lsla_man.

```
trace_mask none dia
```

turns off tracing for all modules except dia_man.

Tracing cannot be turned on for a module that was not included in the trace mask specified in the bindfile with which the core image was created.

FNP BREAKPOINT FACILITY

The control table interpreter in the FNP allows breakpoints to be set in the interpreted control tables. A breakpoint will cause the line encountering it to stop execution in the interpreter until a command is given to restart it.

Breakpoints are often a useful tool but a certain amount of care must be excercised in their use. The following points are important:

1. Breakpoints can only be set in interpreted op blocks. They cannot be set at machine instructions.

2. While at a break, the line is executing an op block equivalent to:

   ```
   wait      0,0,0
   ```

   followed by no status blocks. This means that timers can run out unnoticed, status will be ignored, hangups can be missed, etc. For this reason, it may be difficult to restart a channel after a breakpoint.

3. Breakpoints cannot be set at subroutine levels where waits would be illegal.

4. Breakpoints cannot be set when a restart may execute a waitm op block.

5. Breakpoints cannot be set at a status op block.

6. If a breakpoint is set at a wait op block, it must be reset before the line is restarted. In addition, a breakpoint may not be set at a wait if any channels are currently waiting at that block.

7. Control tables that use local internal variables (as opposed to variables in the TIB extension) cannot depend on these variables being preserved during the break unless no other channels that may use the same control tables are running.

8. No notice is given when a channel encounters a breakpoint. The list_break request will list all breakpoints and show what channels are stopped at each one.

To set a breakpoint:

```
set_break address {channel_name} {-stop_trace}
sb address {channel_name} {-stop_trace}
```

will set a breakpoint at the address specified. If a channel_name is given, the breakpoint will apply to that channel only. Any other channel encountering the breakpoint will continue execution. If -stop_trace is specified, the FNP will automatically suspend tracing if any channel stops at that breakpoint.

To reset a break:

```
reset_break address
reset_break -all
rb address
rb -all
```

will reset a break at the address specified. Any channels stopped at the break are <u>not</u> automatically restarted. If -all is specified, all breaks will be reset.

To start a channel stopped at a breakpoint,

```
start channel_name {address} {-reset} {-start_trace}
start -all
sr channel_name {address} {-reset} {-start_trace}
sr -all
```

will restart the channel specified. If an address is given, the channel will be restarted at the address given, instead of where it was stopped. If -reset is specified, the break will be reset before the channel is started. If -start_trace is specified, tracing will resume as the channel is restarted. If -all is specified, all channels at breakpoints at the time the request is issued will be restarted.

To list FNP breakpoints:

```
list_break
lb
```

will list all FNP breakpoints and the channels stopped at each.

Performance Analysis Requests

The FNP software periodically samples the instruction counter to determine whether the FNP is running or idling. This meter can be displayed with the idle_time request, as follows:

```
idle_time {-reset|-rs}
```

will print the percent of time the FNP has been idling since bootload, or the last time the request was invoked with the -reset control argument.

The sampling interval used by the FNP for metering this data can be printed or set with the following request:

```
sample_time {new_time}
```

where new_time, if specified, is the new sampling interval in milliseconds. The argument must be between 1 and 1000. If no argument is given, the current sampling interval is printed. The default sampling time when the FNP is booted is 50 milliseconds.

More detailed information on FNP usage can be collected by configuring the module 'ic_sampler' in the FNP core image. This module will periodically sample the instruction counter (at the rate set by the sample_time request) and add 1 to a bucket which

represents a small range (typically 16) of FNP addresses. With this data it can be determined with some precision where the FNP is spending its time when it is running.

This instruction counter sampling feature is controlled by the ic_sample request, which is only accepted if the ic_sampler module is configured in the FNP. The following options of the request are used to control ic sampling:

ic_sample start

starts the IC sampling feature. Sampling is normally disabled when the FNP is booted.

ic_sample stop

stops IC sampling.

ic_sample reset

zeroes all the sampling buckets.

The following options are used to display the information collected:

ic_sample module

prints a table showing each module in the core image and what percentage of samples collected occur in that module.

ic_sample histogram¦hist {fraction}

prints a histogram showing each bucket address that has data, and the percent of non-idle time that bucket represents. The fraction argument, if specified, must be a floating point number between 0.0 and 1.0. If this option is used, the histogram will only contain the most frequently used buckets. Enough buckets will be printed so that the fraction specified of the total data collected will be printed. For example, if the fraction is .9, 10% of the data collected will not be displayed by discarding infrequently referenced buckets. This option is useful in deleting "noise" from the histogram.

Other Requests

To select a specfic channel:

line {channel_name}

will locate the TIB, software communications region, and hardware communications region of the channel specified. Once these addresses are set, fields in these control blocks can be referenced by name in any expression in other requests. The channel can be specified either in Multics form (a.h012) or as an FNP channel number (1014). If no channel is specified, the name of the current channel is printed. If the channel selected is not on the current FNP, the proper FNP will be selected automatically.

To print a summary of FNP buffer usage:

buffer_status {-brief¦-bf}
bstat {-brief¦-bf}

will print a table showing each channel and how much buffer space in the FNP it is using. If -brief is used, only summary information is printed.

To set a symbol:

set symbol value

where symbol is '*', 'tib', 'hwcm', 'sfcm', or any user-defined symbol. Setting control block addresses (tib, hwcm, sfcm) is more easily done with the line request, but can be manually done with the set request in case internal FNP tables have been damaged. Note that setting any of these control block addresses has no effect on the current value of other control blocks. Setting "*" is also done by any dump or patch request. Once set, a symbol may be used in any expression in any other request.

To display a list of modules in the core image:

map

will display a list of modules, their addresses, and the dates on which they were last assembled.

To interpret an FNP address:

```
convert_address {address1} ... {addressn}
cva {address1} ... {addressn}
```

will convert each address to any other meaningful form that can be derived. For example, octal values will be converted to module|offset, and vice versa.

To find the explanation of any FNP symbol (usally the output of a flags or convert_address request):

```
explain sym1 {sym2} ... {symn}
```

where symi are symbols to be explained. This command will print the comment from the line in macros.map355 that defines the symbol.

To execute any Multics command:

```
e Command Line
```

will pass 'Command Line' to the command processer.

To exit from debug_fnp,

```
quit
q
```

Summary of debug_fnp Requests

Request	Arguments	Function
block, blk	address {-offset N}{-length N}	display a control block
block_chain, blkc	address {-offset N}{-length N}	display a chain of control blocks
buffer, buf	address {mode}{-brief}	display a buffer
buffer_chain, bufc	address {mode}{-brief}	display a chain of buffers

Request	Arguments	Function
buffer_status, bstat	{-brief}	summarize buffer usage
call_trace	address {-long}	trace subroutine calls
clear_flag	flag_symbol	turn off a flag bit
convert_address, cva	expressions	reinterpret expressions in various formats
display, d	address {length}{mode}	display contents of memory location
dump	path	select an FNP dump
dump_dir	path	select the directory in which dumps are to be found
dumps		print a list of available dumps
e	command_line	execute a Multics command line
explain	symbols	print comments associated with symbol definitions
flags	address {type}	list the names of

Request	Arguments	Function
		flag bits set at an address
	=expression type	interpret the flag bits specified by expression according to type
fnp	fnp_tag	select a running FNP
ic_sample	(various)	control the instruction counter sampling feature
idle_time	{-reset}	print the percentage of FNP idle time
image	path	select a core image
last_dump		select the latest dump in the dump directory
line	channel_name	set control block addresses for specified channel
list_break, lb		list current FNP breakpoints
map		print a list of module names with starting

Request	Arguments	Function
		address and date compiled
next_dump		select the next latest dump
patch	address values	modify one or more memory locations
prev_dump		select the next earliest dump
print_trace	{start}{count}	display a specified portion of the trace buffer
quit, q		exit from debug_fnp
regs		display register contents
reset_break, rb	address or -all	reset breakpoint(s)
sample_time	{interval}	print or change the sampling interval
select_fnp	fnp_tag	select the dump of a specific FNP within a BOS dump
set	symbol value	assign a value to a symbol

Request	Arguments	Function
set_break, sb	address {channel}{-stop_trace}	set a breakpoint
set_flag	flag_symbol	turn on a flag bit
start	channel {address}{-reset} {-start_trace}	restart a breakpoint
start_trace		resume tracing
stop_trace		suspend tracing
trace_mask	{modules}	display or modify the list of modules being traced
what		print the name of the current FNP, dump, or core image
why		print the error message describing a crash

Name: online_dump_355, od_355

The online_dump_355 command formats an FNP dump generated by the BOS command FD355 for output to a terminal or a printer.

Usage

```
online_dump_355 erfno {-control_args}
```

where:

1. erfno
   is the number of the error report form (ERF) associated with the dump.

2. control_args
   may be chosen from the following list of optional control arguments:

   -dev STR
   sends the output to the device named STR, which may be the pathname of a file if file_ is specified with the -dim control argument (below).

   -dim STR
   outputs the dump using the ios_ device interface module (DIM) whose name is STR.

   -tag I
   formats only the dump of the FNP whose tag is I, where I is either a, b, c, or d. If this control argument is omitted, dumps are formatted for all FNPs that were configured when the FD355 command was issued.

Note

If neither the -dim nor the -dev control argument is specified, default values of prtdim and prta, respectively, are supplied. Unless the process issuing the command is sufficiently privileged to attach a printer, these control arguments must be supplied.

Name: online_dump_fnp, od_fnp

The online_dump_fnp command is used to output an ASCII dump of an FNP corresponding to a core dump in >dumps.

Usage

online_dump_fnp {-control_args}

where control_args can be chosen from the following:

-tag FNP_tag
: specifies the FNP tag component of the dump name (see "Note" below).

-date mmddyy, -dt mmddyy
: specifies the date component of the dump name (see "Note" below). If this argument is not supplied, the current date is used.

-time hhmm, -tm hhmm
: specifies the time component of the dump name (see "Note" below).

-pathname path, -pn path
: specifies the pathname (relative or absolute) of the dump segment. This argument, if specified, overrides the -date, -time, and -tag control arguments if any of them are supplied. If this argument is not supplied, the -tag, -date, and -time control arguments are used to find the dump segment (see "Note" below).

-dim dim_name
: specifies the device interface module (DIM) to be used to output the dump. For reasons of compatibility, it must be an IOS-type DIM. This argument must be supplied.

-device device_name, -dv device_name
: specifies the device to which the dump is to be output. This argument must be supplied. If -dim is file_, -device must be a relative or absolute pathname.

Note

The name of an FNP dump segment is of the form fnp.TAG.DATE.TIME, where TAG, DATE, and TIME are as described above under the relevant control arguments. In specifying the dump segment to online_dump_fnp (other than by using the -pathname control argument), the TAG and/or TIME component may be omitted if the remaining information is sufficient to uniquely identify the dump.

Name: tty_analyze, tta

The tty_analyze command analyzes the contents of a copy of tty_buf which it extracts from a dump. It performs certain tests to verify the consistency of the contents of tty_buf.

Usage

tty_analyze erfno {-control_arg}

where:

1. erfno
   is the number of the dump from which the copy of tty_buf is to be extracted.

2. control_arg
   may be -long (or -lg). If present the contents of each input and output buffer will be printed, otherwise only the addresses of the buffers will be printed.

APPENDIX C

FNP MEMORY CONFIGURATOR

This appendix provides a memory configurator that can be used to approximate maximum memory utilization in the FNP.

MULTICS COMMUNICATION SYSTEM MEMORY CONFIGURATOR

TABLE 1

MODULE	LENGTH		MUST HAVE	TOTAL
acu_tables	112			
ards_tables	710			
autobaud_tables	262			
breakpoint_man	268			
bsc_tables	2296			
console_man	476		*	476
g115_tables	1920			
ibm3270_tables	616			
t202_tables	546			
trace	446			
trace buffer	xxx			
vip_tables	1152			
hsla_man	3340			
lsla_man	1936			
control_tables	1674		*	1674
dia_man	3136		*	3136
interpreter	1918		*	1918
scheduler	1178		*	1178
utilities	1630		*	1630
interupt vect	512		*	512
iom tables	32		*	32
hsla hwcm	xxxx	No. of HSLAs X 512		
init	3278		*	----
			SubTotal1	

TABLE 2

HSLA Chans X (Half/Full duplex or Echo-mode *) = (Half/Full duplex or Echo-mode *)

HSLA Chans	Half/ Full duplex	Echo-mode *	Half/ Full duplex	Echo-mode *		TOTAL
32 tty ≤9600	216(1)	248(2)	6912	7936		
40 tty ≤9600	216	248	8640	9920		
48 tty ≤9600	"	"	10368	11904		
56 tty ≤9600	"	"	12096	13888		
64 tty ≤9600	"	"	13824	15872		
72 tty ≤9600	"	"	15552	17856		
80 tty ≤9600	"	"	17280	19840		
88 tty ≤9600	"	"	19008	21824		
96 tty ≤9600	"	"	20736	21328		
				SubTotal2		

T A B L E 3

LSLA Chans X ----- or ----- = ----- or -----

	Half/ Full duplex	Echo- mode *	Half/ Full duplex	Echo- mode *		TOTAL
No. of tty $\le$300	00(5)	132(6)				
No. of LSLAs $\le$6	170(7)	170(7)				
				SubTotal3		

* echomode = echoplex,crecho,lfecho, and/or tabecho

T A B L E 4

No. of Chans X ----- = TOTAL

(IO buf + Misc.) = -----

g115 protocol	192	108(3)		300		
2780 protocol	224	118(4)		342		
3780 protocol	256	118		374		
				Subtotal4		

Legend: (1) =

SFCM	52
TIB	34
TB tbl	2
IO buf	128
	216

Legend: (2) =

SFCM	52
TIB	34
TB tbl	2
IO buf	160
	248

Legend: (3) =

SFCM	52
TIB	34
TB tbl	2
TB ext	20
	108

Legend: (4) =

SFCM	52
TIB	34
TB tbl	2
TB ext	30
	118

Legend: (5) =

TIB	34
TB tbl	2
IO buf	64

	100

Legend: (6) =

TIB	34
TB tbl	2
IO buf	96

	132

```
Legend:  (7)  =     SFCM     42
                  IO buf    128
                           ----
                            170
```

HOW TO USE THE CONFIGURATOR

To compute the available free buffer space (as a result of configuring required software modules) use the following algorithm.

1. Sum the length of all required modules in the TOTAL column (Table 1) to obtain SubTotal1.
2. Enter the appropriate length of Half/full duplex, or echoplex HSLA tty channels in the TOTAL column (Table 2) to obtain SubTotal2.
3. Enter the appropriate length of Half/Full duplex, or echoplex LSLA tty channels in the TOTAL column (Table 3). Next, multiply the number of configured LSLAs by 170 and enter in TOTAL column and add to obtain SubTotal3.
4. Enter the appropriate length of the g115 and 2780/3780 channels in the TOTAL column (Table 4) to obtain SubTotal4.
5. Add SubTotal1, SubTotal2, SubTotal3 and SubTotal4. Subtract that sum from the value 32,768; the difference is the amount of available free buffer space.

APPENDIX D

LAYOUT OF FNP MEMORY

This appendix describes the assignment of various parts of FNP memory under Multics Communication System. This information may prove helpful in reading dumps of the FNP.

The low-order 1000 (octal) locations contain the fixed fields described below. Locations 640 through 775 are not all used at present; the unused locations are reserved for future extensions of the system communications region.

Location	Name	Description
0- 377	intv	IOM interrupt vectors
400- 417	intv	IOM interrupt cells
		IOM Channel Fault Status Words
420	tyfts	typewriter fault status word
421	crfts	card reader fault status word
422	lpfts	line printer fault status word
423	mtfts	magnetic tape fault status word
424	difts	DIA fault status word
425		
426	h1fts	hsla 1 fault status word
427	h2fts	hsla 2 fault status word
430	h3fts	hsla 3 fault status word
431	l1fts	lsla 1 fault status word
432	l2fts	lsla 2 fault status word
433	l3fts	lsla 3 fault status word
434	l4fts	lsla 4 fault status word
435	l5fts	lsla 5 fault status word
436	l6fts	lsla 6 fault status word
437	tmfts	timer fault status word

CPU Fault Vectors

440	suflt	startup fault
441	sdflt	shutdown fault
442	parflt	memory parity fault
443	iopflt	illegal operation fault
444	ovflt	overflow fault
445	memflt	illegal memory operation fault
446	dvflt	divide check fault
447	ipiflt	illegal program interrupt fault

IOM Mailbox Communications Region

450	itmb	interval timer mailbox
451	etmb	elapsed timer mailbox
452		
453		
454- 455	dimb	DIA pcw mailbox
456- 457	dist	DIA status icw mailbox
460- 461	tyst	typewriter status icw mailbox
462- 463	tyicw	typewriter data icw mailbox
464- 465	crst	card reader status icw mailbox
466- 467	cricw	card reader data icw mailbox
470- 471	lpst	line printer status icw mailbox
472- 473	lpicw	line printer data icw mailbox
474- 477		
500- 517	l1mb	lsla 1 hardware communications region
520- 537	l2mb	lsla 2 hardware communications region
540- 557	l3mb	lsla 3 hardware communications region
560- 577	l4mb	lsla 4 hardware communications region
600- 617	l5mb	lsla 5 hardware communications region
620- 637	l6mb	lsla 6 hardware communications region
640- 775		Communications region (described in Section 10)
776- 777		missing module code

Starting at location 1000(8) are the HSLA hardware communications regions for as many HSLAs as the core image supports. Each hardware communications region occupies 20(8) words, and there is one for each subchannel on a given HSLA; thus the hardware communications regions for each HSLA occupy 1000 locations. These regions are described in Section 10. The number of HSLAs supported is determined by the "hsla" statement in the core image bindfile, and is kept in .crnhs (location 654) in the system communications region.

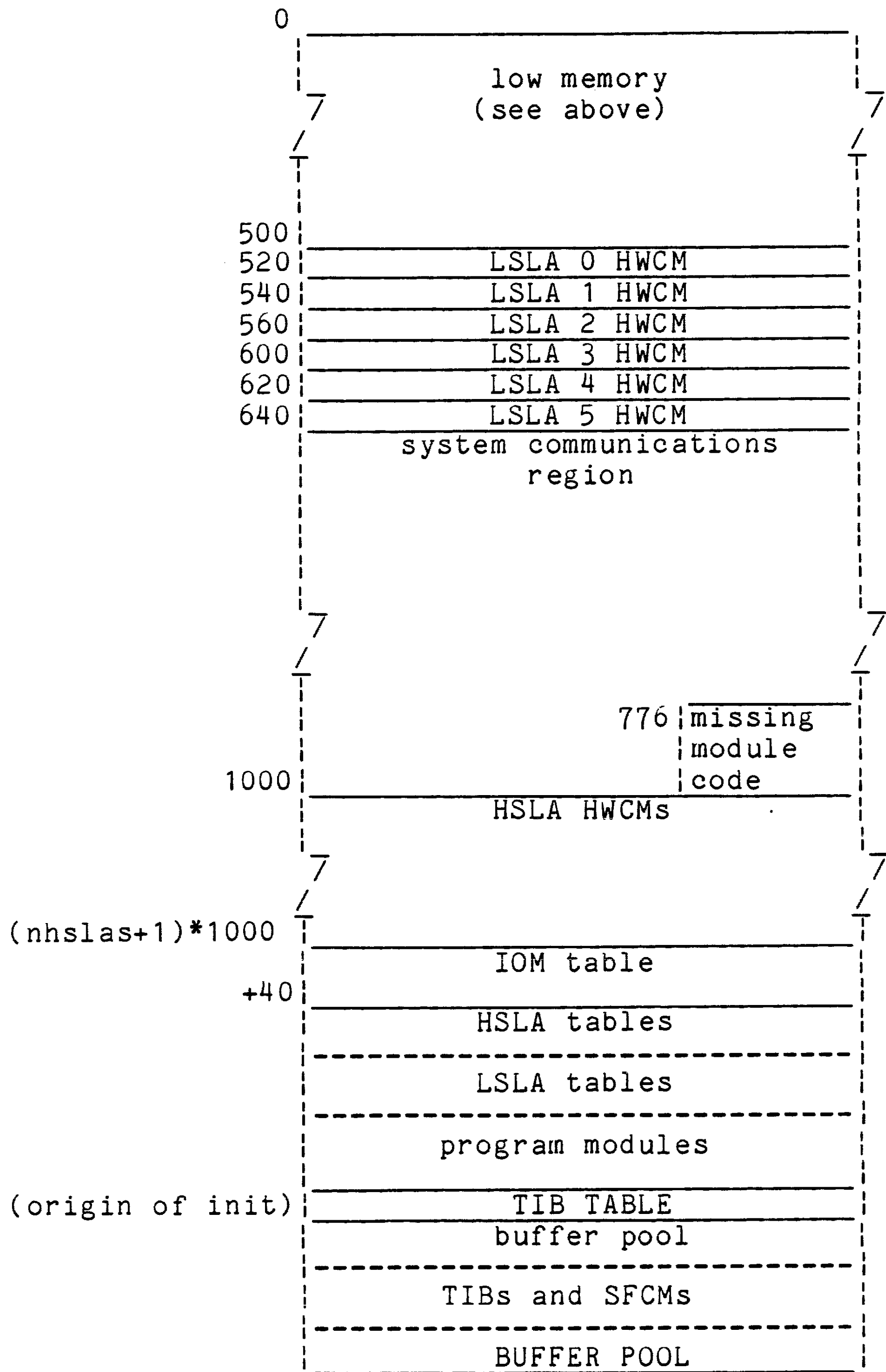
Following the HSLA hardware communications regions is the IOM table (described in Section 10); its starting address is either 1000, 2000, 3000, or 4000, according to whether zero, one, two, or three HSLAs are supported. This address is kept in .criom (location 653) in the system communications region.

Next come the HSLA tables for each configured HSLA, followed by the LSLA tables for each configured LSLA. The addresses of the individual tables are found in the second word of the IOM table entry for the corresponding adapter. Each HSLA table is 100(8) words, two for each possible subchannel. The effective size of each LSLA table depends on how many time slots are in use; an entry of all ones marks the end of an LSLA table.

After the LSLA tables come the FNP programs themselves, in the order specified by the bindfile. The beginning of the program area can be identified in a dump by the first appearance of a module name after the absolute address. Note that the trace table comes at the end of the trace module, as described in Section 14.

As explained in Section 15, the last module, init, is converted to buffer space after initialization; accordingly, after the TIB table at the beginning of init, the remainder of the module is either free or in use as buffers. Immediately following the location that was originally the end of init come the TIBs and HSLA software communications regions, allocated by init as described in Section 15; the remainder of memory after the last TIB is again part of the buffer pool. Note that in the dump the entire high-order portion of memory is marked as being part of init.

The following is a schematic drawing of all FNP memory.



APPENDIX E

AUTOMATIC BAUD RATE DETECTION

The automatic baud rate detection (autobaud) feature of the Multics Communication System is designed to recognize/configure the baud rate (bits per second) of an asynchronous HSLA channel at dialup time. The autobaud facility selects a baud rate of 1200 if a lead from the answer modem is on; otherwise it selects a rate of 110, 133, 150 or 300 baud based on the sampling of bit changes for the first incoming character, which is expected to be "l" or "L".

LEAD CONTROL SELECTION OF 1200 BAUD

If the answer modem turns on pin 12 of the cable connected from it to the FNP, the channel is set to 1200 baud and the channel is then handled in the normal manner. The answering modem can be set to respond to a switch on the originate modem which indicates that the terminal is operating at 1200 baud. The operation of the channel will not appear any different to the user than if he had dialed into a strictly 1200 baud channel (there is no requirement for the user to type any characters before receiving the login banner).

BIT SAMPLING SELECTION OF OTHER BAUD RATES

If the signal on pin 12 is off, sampling for the bit changes of an input character is performed. To accomplish this, the following sequence occurs:

1. The user establishes a connection with the host.

2. The user types in either the letter "l" or "L".

3. The software in the FNP scans the incoming bit stream looking for bit changes at 300 baud. Since the "l" (or "L") character is known, the changes in state of the bits ("0" or "1") indicate the timing necessary to transmit the bits, and therefore the baud rate of the channel is determined.

4. From here the channel is handled in the normal manner. The answerback is checked, initial string sent if any and the login banner is displayed.

5. The user types in any of the preaccess commands (MAP, etc.) if desired.

6. The user logs into the system using "l" again for "login", enter, etc.

MODEMS

Any asynchronous HSLA channel may be configured with the autobaud feature. For dialup channels not using special modems, the channel will only be able to detect baud rates up to 300 baud. In this range, most modems and acoustic couplers are able to interface with host modems. For hardwired channels, a special switch could be installed to make use of the pin 12 lead change and operate up to 1200 baud.

There are several modems which can be used on channels configured with the autobaud feature which make use of the pin 12 lead change. Vadic (3467) and Western Electric (212A) are two manufacturers that make such modems. Special circuits are required to handle the data over voice-grade dialup lines. It is because of these special circuits that a modem of one manufacturer will not necessarily be able to interface to a modem of another manufacturer.

Modem Options Needed for Autobaud

The modem must be able to indicate the position of the high speed switch on the originate modem. This indication is signaled by pin 12 of the RS232 interface.

a. In the Western Electric 212A modem, this feature is enabled in the answer modem when the "YQ" option is installed.

b. It is currently unknown how other modem manufactures treat this feature.

ALGORITHM

The following is the algorithm used to determine the baud rates for various terminals. This discussion is directed towards people knowledgeable in communications who wish to understand the design of the autobaud facility.

egend:

```
B - start bit (begin)
P - parity bit
E - stop bit (end)
X - bit in the 300 baud sample whose value is uncertain because
    it depends on the parity bit of incoming character
```

otes

1. A communications line can be in one of two states, 1 (also called "mark" condition) and 0 (also called "space" condition).

2. The line is normally held in a 1 state.

3. The first incoming bit is always a start bit, which (being a 0 bit by definition) changes the line state to 0 and causes the hardware to begin sampling bits.

4. The bits of a charcter are transmitted after the start bit starting with the least significant bit and ending with the parity bit after the most significant bit, followed by one or two stop bits.

5. A stop bit is a 1 state held on the line for one bit time interval.

6. The channel is set up to receive 7 data bits plus 1 parity bit (8 information bits) with one start and stop bit (a total of 10 bits). The parity bit in the sampling is stripped off before the comparison.

7. Bits in the sampling dependent on the parity bit of the incoming character are masked off (shown as X in sampling lines below).

In the following diagrams, all sampling is done at 300 baud. n the incoming lines below, the time between the vertical bars s the bit time for the indicated baud rate in relation to the 00 baud rate shown in the sampling lines.

```
f the terminal is operating at 110 baud:
ncoming "L" is:  |  B  |  0  |  0  |  1  |  1  |  0  |  0  |  1  |  P  |  E  |
ampling yields:  |B|0|0|0|0|0|0|0|P|E|        |B|0|0|0|0|0|1|1|P|E|
                   first char= 000              second char= 140

ncoming "l" is:  |  B  |  0  |  0  |  1  |  1  |  0  |  1  |  1  |  P  |  E  |
ampling yields:  |B|0|0|0|0|0|0|0|P|E|        |B|0|0|1|1|1|1|1|P|E|
                   first char= 000              second char= 174
```

```
If the terminal is operating at 150 baud:
incoming "L" is:   ¦ B ¦ 0 ¦ 0 ¦ 1 ¦ 1 ¦ 0 ¦ 0 ¦ 1 ¦ P ¦ E ¦
sampling yields:   ¦B¦0¦0¦0¦0¦0¦1¦1¦P¦E¦B¦0¦0¦0¦1¦1¦X¦X¦P¦E¦
                     first char= 140     second char= 030 (170 if P=1)

incoming "l" is:   ¦ B ¦ 0 ¦ 0 ¦ 1 ¦ 1 ¦ 0 ¦ 1 ¦ 1 ¦ P ¦ E ¦
sampling yields:   ¦B¦0¦0¦0¦0¦0¦1¦1¦P¦E¦B¦0¦1¦1¦1¦1¦X¦X¦P¦E¦
                     first char= 140     second char= 036 (176 if P=1)
```

If the terminal is operating at 133 baud:

The following is an approximation of what can occur. It has been found that the bit stream seen when sampling a 133 baud "l" at 300 baud is dependent on the hardware involved. This includes the terminal as well as the channel doing the sampling. The characters in parentheses and braces are the result of these variations and were for the most part arrived at empirically.

There are no uppercase characters on currently known 133 baud devices. When the shift key is depressed, a "shift-up" character is sent. When the shift key is released, a "shift-down" character is sent.

Note that the stop bit for the first 300 baud character is shown with a shorter bit time. This is an attempt to explain what the different channel hardware reports "seeing" during the sampling. This may be due to the lack of a proper "1" state on the line at the time the stop bit is expected.

```
For an EBCD terminal (l=61):
first 4 Bits of incoming "l" is:   ¦   B   ¦   1   ¦   0   ¦   0   ¦...
sampling yields:                   ¦ B ¦ 0 ¦ 1 ¦ 1 ¦ 1 ¦ 0 ¦ 0 ¦ 0 ¦ P ¦...
                                   first char= 016  {030, 034}

last bits of incoming "l" is:      ¦   0   ¦   1   ¦   1   ¦   0   ¦
continued sampling yields:         ¦E¦ B ¦ 0 ¦ 1 ¦ 1 ¦ 1 ¦ 1 ¦ 1 ¦ 0 ¦ P ¦
                                   second char= 076 (074)  {170}
```

Therefore, if the first character is a 016 and the second character is a 076 or 074, the channel is set to 133 baud.

It has been observed that if the first character is 030 or 034 and the second is 170 (those noted in braces above), the channel should also be set to 133 baud.

```
For a Correspondence terminal (l=06):
First 4 Bits of incoming "l" is:  |    B    |    0    |    1    |    1    |...
Sampling yields:                  | B | 0 | 0 | 0 | ? | 1 | 1 | 1 | P |...
                                  first char= 160 (170)

Last bits of incoming "l" is:     |    0    |    0    |    0    |    1    |
Continued sampling yields:        |E| B | 0 | 0 | 0 | 0 | 0 | 1 | 1 | P |
                                  second char= 140 (100)
```

Therefore, if the first character is a 160 or 170 and the second character is a 140 or 100, the channel is set to 133 baud.

INDEX

A

accept input 13-6, 13-7, 13-8

accept_direct_input operation code 4-4

accept_direct_output operation code 4-5, 4-6

ACU
  see automatic call unit

addressing strings 12-5

answer table 8-2

answerback 12-2, 12-47, A-11

answering service 1-4, 3-14, 3-22, 7-2

autobaud E-1

automatic baud_rate detection 12-1

automatic call unit (ACU) 3-13, 12-1, 12-7, A-3

B

base address word (BAW) 13-17, 13-18

baud rate 5-5, 12-5, 12-29, 15-6, A-7, E-1

BAW
  see base address word

binary synchronous 12-2, 12-7, 12-50

bindfile 12-43, 14-5, 14-6, 15-1, 15-5

block check 12-41, 12-43

bootload communications region 15-2, 15-3, 15-4

bootload program 15-2

break character 1-5, 4-4, 5-8, 5-9, 5-10, 5-15, 5-16, 5-18, 5-22, 9-4, 12-7, 12-49, 13-19, A-7

break list 12-4, 13-14

breakall mode 5-16, 5-24

breakpoint 5-24, 12-38, A-4, B-11

buffer 5-2, 5-5, 6-4, 10-7, 10-8
  allocation 16-4
  allocation and copying 5-2
  chain 6-3, 6-4
  pool 10-7, 15-5
  size 5-5

C

canonical form 5-8

canonicalization 5-9, 5-11

CCT
  see character control table

CCT descriptor 13-18

CDT
  see channel definition table

central system (CS)
  responsibilities 1-1

channel
  initialization 7-2
  lock 4-7, 6-1, 6-2
  name 2-7, 5-17
  output 1-5
  type 3-2

channel definition table (CDT) 1-4, 2-3, 3-1, 3-21, 7-2, 10-5, 12-43, 15-2, 15-6

channel master file (CMF) 12-43, 12-46

channel_manager 1-4, 3-3, 6-1
  $check_modes entry 3-7
  $control entry 3-6

channel_manager (cont)
  $get_modes entry 3-10
  $interrupt entry 3-10
  $read entry 3-4
  $set_modes entry 3-9
  $write entry 3-5

character control table (CCT) 12-31, 12-45, 13-16, 13-17

checksum 2-2

circular buffer 2-3, 4-4, 4-7, 7-1, 8-2, 13-8, A-2, A-6

circular queue 2-3

cmtv 3-2, 3-20

command data 4-1, 4-5

communications channels 1-1

concentrator 3-1
  channel 1-2

configuration
  cards 7-1
  deck 2-2, 2-3, 7-1

control blocks
  per-channel 2-5

control operation 3-6, 5-19

control table 5-24, 9-3, 9-4, 12-1, 13-5, 13-13, 13-14, 13-15, 13-17

conversion 5-8, 5-11
  and translation tables 2-7
  table 5-18, 5-21, 6-5

copying 5-10

core image 9-2, 12-1, 12-43, 15-1, 15-2, 16-3, 17-1

coreload command 17-2

crash 8-6

crash interrupt 1-6

crawlout 6-2

D

data buffer 2-5

database pointer 3-4

debug_fnp command 5-24

delay 13-15
  characters 13-19

delay (cont)
  queue 2-6, 4-4, 4-5, 4-7, 6-1, 6-2, 8-4, 8-5
  table 5-4, 5-21, 10-10, A-5

device
  info tables 12-2

device element 10-2

device index (devx) 2-4, 2-7, 3-1, 3-4, 3-20, 4-5, 5-2, 5-7, 5-17, 5-18

device speed codes 10-7

devx
  see device index

DIA
  see direct interface adapter

dialout operation 3-13

dialups 1-4

direct interface adapter (DIA) 4-1, 6-5, 13-1
  initialization 15-5
  interrupts 13-5
  lock 13-3, 13-5
  request queue 10-7, 10-9, 13-4, 13-6, 13-8

disable_breakall_mode operation 5-24

dn355 1-2, 3-3, 4-1

dn355_data 2-1

dn355_mailbox 2-1

dn355_util 6-1, 6-5

dump 14-6

dump_fnp operation 5-23

E

echo buffer 10-8, 12-31, 13-12, 13-14, 13-15, 13-19, 13-20

echo negotiation 5-16

elapsed timer 11-7

enable_breakall_mode operation 5-24

erase 5-15

erase character 5-13, 5-14

erase processing 5-14

error message queue 10-10

error messages 8-1, 8-6

escape 5-15
  character 5-3, 5-14, 5-15
  sequence 5-2, 5-4, 5-8, 5-15

event channel 5-18

exhaust status 12-8, 12-49, 13-14

F

fault 16-1, 16-3, 16-5, B-9

fault vector 14-2

fill character 13-10, 13-11, 13-12, 13-13, 15-7

FNP
  bootload 15-2
  clock 11-5
  console 13-20
  core images 17-1
  crashes 8-6
  dump 16-1, 17-1
  dump analysis 17-3
  initiated transactions 4-2
  interface modules 1-2
    see module
  space management 14-1

fnp_break operation 5-24

fnp_info 2-1, 2-6, 4-7, 7-2

fnp_multiplexer 1-2, 1-4, 3-3

formatting 5-2

formfeed 4-4, 4-5, 5-4

free
  block 2-4, 10-7, 10-8, 14-1
  chain 10-8
  pool 7-1, 14-1
  space 2-3, 4-3
  space errors 8-7

G

get_buffer entry 6-3

get_chars operation 5-7

get_line operation 5-7

global operations 4-6

H

hangup 1-6

hardware communications region 10-3, 13-15, B-5, B-15

hardware communications region (HSLA) 15-3, 15-6

hardware communications region (LSLA) 14-4, 15-7

hardware status queue 13-15

high-speed line adapter (HSLA) 7-2, 13-15, E-1
  initialization 15-6
  software communications region 10-4
  table 10-5, 13-15, 14-2, 15-3, 15-5

hndlquit mode 1-5

HSLA
  see high-speed line adapter

I

I/O command 2-2

I/O module 12-49, 12-50

I/O system (iox_) 5-1

idle time metering 11-7

indicator 5-4, 5-15, 6-5

indirect control words (ICWs) 10-3

initialization
  answering service 1-4
  channel 1-4

input chain 2-5, 3-5, 3-12, 4-4, 5-7, 5-19, 5-20, 8-3, 12-26, 12-30

input conversion table 5-15

input frame 13-9, 13-10, 13-11, 13-13, 14-4, 15-6

input message 12-27

input_accepted operation code 4-4

input_in_mailbox operation code 4-3

instruction counter sampling 11-7

interface modules
  FNP 1-2
  user 1-2

interpreter 9-4, 12-5, 13-5, 13-13, 16-4

interrupt 3-3, 3-10, 3-23, 4-1, 9-3
  cells 11-1
  handler 3-3, 3-10, 3-21

interrupt (cont)
  level 4-2, 4-3
  type 3-3, 3-11
  vector 11-1, 11-2, 15-5, 15-7

interval timer 11-5

IOM
  channel fault 14-2, 14-3, 16-1
  table 10-1, 14-2, 15-3, 15-5
  table entry 10-4

J

jump table 11-1, 11-2

K

keyboard addressing 12-3, 12-17, A-10

kill character 5-13, 5-14, 5-15

kill processing 5-13, 5-14

L

LCNT
  see logical channel name table

LCT
  see logical channel table

LCT entries (LCTEs) 2-3

LCTE 2-6, 4-7, 5-2, 5-7, 5-18, 6-1, 6-3, 7-2, 8-2

line
  adapter 7-2
  control 12-49, 12-50, A-5
  number 4-5, 13-6, 14-2
  status 12-37, 12-49, 12-50, A-9
  type 5-8, 12-1, 12-5, 12-13, 12-19, 12-21, 12-31, 12-46, 15-7, A-7

line_control order 12-37

local variable 12-32, 12-33, 12-34, 12-35, 12-38, 12-39

lock errors 8-7

locking 6-1

lock_channel entry 6-1

lock_channel_int entry 6-1

logical channel 3-1, 3-10

logical channel name table (LCNT) 2-4, 2-6, 5-18, 7-2, 8-2, 8-4

logical channel table (LCT) 2-3, 3-1, 6-4, 7-2

low-speed line adapter (LSLA) 7-2, 13-9
  initialization 15-6
  interrupt processor 13-10
  software communications regions 10-4
  table 10-4, 13-9, 13-12, 14-2, 15-3, 15-5, 15-6
  time slots 10-4

LSLA
  see low-speed line adapter

M

mailbox 13-1
  area 2-1, 4-1
  header 8-6, 14-3, 15-8, 16-1
  operation code 4-1, 4-7
  queue 13-4, 13-6

major channel 2-4, 3-1, 3-3, 3-16, 5-19

marker status 12-8, 12-9, 13-19

master dispatcher 11-1, 11-3, 11-7, 15-7

metering 8-1, 14-3

modem E-1

modem option E-2

modes 3-7, 5-8

modes operation 5-19, 5-22

module
  call-switching 1-4
  chain 16-3
  FNP interface 1-2
  FNP multiplexer 1-2
  multiplexer 1-4, 2-4, 3-5
  number 14-6
  transfer vector 3-20
  tty_index 1-3
  tty_read 1-3
  tty_write 1-3
  user interface 1-2, 1-3

Multics processes 1-1

multiplexed channel 3-1

multiplexer 3-1
  database 2-1, 3-16, 8-4
  initialization 3-14, 3-21, 7-2
  loading 3-14, 3-22
  module 2-4, 3-1, 3-2, 5-19, 7-2, 12-49
    see module

multiplexer (cont)
  subchannel 2-4, 3-1
  termination 3-16
  transfer vector module 3-20
  type 2-4, 3-2, 3-20, 8-2, 8-4

multiplexing 1-2

multiplexing interfaces 1-4

N

newline 5-4, 5-7, 5-8

nonmultiplexed channel 2-5, 5-17

O

op block 12-5, 12-10, B-5

operation code (opcode) 2-2, 4-5, 10-9, A-1

output chain 2-5, 2-6, 3-5, 4-4, 4-5, 5-5, 5-6, 5-19, 5-20, 8-3, 12-26, 12-27, 14-4

output conversion 5-2

output frame 13-9, 13-10, 13-12, 15-6

output message 12-27, 12-28

owning process 2-5

P

parent multiplexer 2-4, 3-1, 3-17, 5-5, 8-2

patch_fnp operation 5-24

PCB
  see physical channel block

PCW
  see peripheral control word

peripheral control word (PCW) 2-1, 6-5

physical channel 2-4, 3-1, 3-3, 8-2, 8-4

physical channel block (PCB) 2-6, 4-7

pre-exhaust status 12-8, 12-49

preliminary conversion 5-2

printer addressing 12-3, 12-17, A-10

printer_off operation 5-20, 5-22

printer_off sequence 5-20

printer_on operation 5-21

priv_channel_manager 1-4, 3-3, 3-14
  $get_devx entry 3-20
  $hpriv_control entry 3-18
  $init_channel entry 3-19
  $init_multiplexer entry 3-14
  $priv_control entry 3-18
  $shutdown entry 3-17
  $start entry 3-16
  $stop entry 3-17
  $terminate_channel entry 3-19
  $terminate_multiplexer entry 3-16

pseudo-DCW 4-5, 8-5, 9-4, A-6

pseudo-DCW list 2-6, 13-5

Q

queue lock 6-2

quit 5-10

quit condition 1-5

R

RCD
  see read control data

RCI 12-50

read control data (RCD) 2-2, 4-2, 13-4
  command 13-6

read text (RTX) 2-2, 4-4, 13-8

read_status operation 5-20

reject request operation code 4-3, 4-4

rejected request 13-8

replay chain 12-30, 12-31

responsibilities
  central system 1-1
  Multics Communication System
    character transmission 1-1

RTX
  see read text

S

scan 12-26, 12-39, 12-40

scan control string 12-39

scheduler queues 10-7

secondary dispatcher 11-3

segment dump 8-4

send_output 3-5
  interrupt 3-21, 4-4, 4-6, 5-2
  operation code 4-3, 4-5

set_terminal_data operation 5-21

size code 2-5, 6-3, 14-1

software communications region 10-4, 13-15, 15-5, 15-7, 16-6, B-5, B-15

software communications region (HSLA) 10-4, 10-7, 13-18, 15-6

software communications region (LSLA) 10-4, 13-10

software status queue 13-15, 13-17

space allocation 6-3

space management 5-9, 6-3

space_needed 6-4

special table 5-21

special_characters table 5-4, 5-20

special_chars 5-15

status 12-7, 12-14, 12-15, B-11

subchannel 2-4
  multiplexer 2-4

subchannel number 2-6, 3-4

submailbox 2-1, 4-1, 9-4, A-1

system communications region 10-1, 10-10, 14-1, 14-2, 14-5, 14-6, 15-5, B-5

system crashes 8-1

T

table descriptor 2-7, 2-8

table types 2-7

tally 2-5

TCB
  see terminal control block

terminal control block (TCB) 2-6, 2-7, 5-2, 5-17

terminal I/O
  buffer space 1-2

terminal information block (TIB) 9-3, 10-6, 10-7, 11-5, 12-3, 12-5, 13-5, 13-15, 14-2, 15-5, B-15
  extension 12-10, 12-22, 12-23, 12-27, 12-32, 12-37, 12-39, 12-40, 12-42, 12-43, B-12
  initialization 15-7
  table 10-7, 13-4

terminal type table (TTT) 5-21

terminate interrupt multiplex word (timw) 2-1, 4-2, 4-4, 4-5, 4-6, 4-7, 13-5, 13-7

terminate status 12-8, 12-9, 12-48

TIB
  see terminal information block

time slot 13-9, 15-3, 15-6

timeout 12-6, 12-48

timer 12-14, 12-18, 12-35, B-11

timer management 11-4

TIMW
  see terminate interrupt multiplex word

trace
  buffer 14-5, 16-3
  mask 14-6, B-10
  table 16-3, 16-4, 16-6, B-10

tracing 14-5

transaction control word 13-3, 13-5, 13-8

transfer timing error 12-49

translation 5-2

translation table 5-11, 5-21, 6-5

tty_ 5-1

tty_analyze command 8-1, 8-4, 8-5, 8-6

tty_area 2-6

tty_attach 5-17

tty_buf 2-2, 4-3, 5-5, 6-3
  header 2-3
  lock 6-2

tty_canon 5-11, 5-12

tty_dump command 8-1, 8-2

tty_event 5-18

tty_index 1-3, 3-3, 5-1, 5-2, 5-17, 5-18, 5-19, 6-1

tty_interrupt 2-5, 3-3, 3-21, 4-3, 4-4, 5-16

tty_lock 4-7, 6-1

tty_meters command 2-3, 8-1

tty_order 5-21

tty_read 1-3, 3-3, 5-1, 5-7, 5-8, 5-16, 5-17, 5-20, 6-1, 6-5

tty_space_man 2-3, 2-4, 6-1, 8-6

tty_state 5-19

tty_tables 2-7

tty_tables_mgr 2-7

tty_util_ 6-1, 6-5

tty_write 1-3, 3-3, 5-1, 5-2, 5-3, 5-4, 5-5, 5-7, 5-17, 5-21, 6-1, 6-5

U

user interface modules
  see module

user process 2-5

V

vertical tab 5-4

W

wakeup 1-3, 1-5, 1-6, 3-3, 3-22, 4-3, 4-4, 4-6, 5-2, 5-7, 5-8, 5-16, 5-18, 6-4

WCD
  see write control data

white space 5-4

wired terminal control block (WTCB) 2-5

write control data (WCD) 2-2, 4-5, 12-6, 13-5, 13-8

write text (WTX) 2-2, 4-6, 13-5

write_status operation 5-20

wru operation 5-22

WTCB
  see wired terminal control block

WTX
  see write text