Remote Terminal Access to a VAX/VMS System in a Heterogenous Network System

---

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston - University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Shoou Jiah, Yiu

August, 1987

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude to my advisor, Dr. Bahaa El-Din for his invaluable assistance, guidance as well as patience for this endeavor. I am also grateful to my thesis committee members: Dr. Ramez Elmasri and Dr. Markenscoff for their invaluable remarks and suggestions.

Also, I wish to thank my fellow graduate students, Janece Partridge and Shia-Chung Teng, for assisting me in many tedious programming sessions.

Last but not least, I would like to dedicate this thesis to my parents, Yiu Hong and Kam May Wah, who have provided me with encouragement and support throughout these years. Also, I would like to thank my sister and my brother and his wife for their invaluable support.

Remote Terminal Access
to a VAX/VMS System
in a Heterogenous Network System.

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston - University Park.

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Shoou Jiah, Yiu

August, 1987

ABSTRACT

In a heterogenous network system consisting of computers of different makes, there is a need for remote terminal access and internet file transfer. The topic of the thesis concentrates on the design and implementation of remote terminal access from any computer to a VAX/VMS system. The only prerequisite condition is that the computer - the remote node - which accesses the VAX/VMS node - the host node - be on the same heterogenous network system based on the ETHERNET standard.

The strength of this design is a fully VMS compatible terminal driver that provides all the screen-oriented capabilities of a DEC VAX/VMS Terminal Driver. Also, its design involves terminal server functions to manage the total number of remote logons, and provides clean recovery from different cases of remote access failure.

The design concentrates the bulk of the code on the VAX/VMS system, thereby allowing relatively easy implementation of remote terminal access on any remote node. The remote node need only to implement a terminal emulator and necessary modules to facilitate the same internet communication protocol as this implementation.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Declining computer hardware costs, coupled with the increasing power of the computer, have led to the increase in the number of systems that are used at a single site. But while every system on site has its designated task, the need to exchange data, share resources, share CPU loads, amongst other distributed applications has led to the interconnection of these systems. Such an interconnection of systems, called a local computer network, has become almost a prerequisite at many sites.

1.1 Motivation and Objectives

In a heterogenous network system consisting of computers of different makes, there is the need for remote terminal access - a form of resource sharing. To put it simply, remote terminal access is the feature by which a user at the remote node can access the host node via the network as if his terminal were directly connected to the host system. The subject of this thesis concentrates on the design and implementation of remote terminal access from any computer to a VAX/VMS system. The only prerequisite condition is that the computer - the remote node - which accesses the VAX/VMS node - the host node - be on the same heterogenous network system based on the ETHERNET standard.

In achieving Remote Terminal Access between a pair of heterogenous nodes, we are really covering different aspects by:

(1) demonstrating the Layered Approach in Network Architecture. At the Presentation Layer, the various Terminal Servers, Terminal Emulator and other routines achieve end-to-end communication by utilizing the underlying common internet Inter-Process Communication (IPC) interface. The IPC, which represents the the Transport Layer, in turn, utilizes the Ethernet layer interface for network communication. The Ethernet layer, encompassing the Datalink and Physical layer, provides the primitives that perform the ultimate physical communication via the ETHERNET coaxial cable that interconnects the various computer systems together.

(2) explaining the need for a virtual terminal driver, and explaining the operating principles involved in

implementing the driver. Such principles cover the interrupt-dispatching mechanisms, the I/O operations on a device, and the synchronization aspects of using both the interrupt-priority levels and the monitor construct of the terminal driver.

(3) demonstrating the generic approach in the design of remote terminal access. This approach concentrates on using basic programming constructs while avoiding system-dependent features. Such a method extends the range of computers that can successfully re-implement the server and emulator routines. Ultimately, the objective is to provide a clear and easily implementable terminal emulator and terminal server at remote node.

1.2 Environment

The heterogenous local network system on Ethernet at the Department of Computer Science, University of Houston, presently consists of VAX/VMS (1 VAX11/780, 4 VAX11/730s), 3B/UNIX version 5 (24 3B2s, 2 3B20s), and Intel/PLM (1 Intel) nodes. At the present, the Inter-Process Communication software (IPC), an in-house project, is being developed on all heterogenous nodes. The IPC encompasses a network transport layer and utilizes the ETHERNET layer common to all nodes.

1.3 Overview of the Thesis

This thesis discusses the design and implementation of a virtual terminal service from any system to a VAX/VMS system on the same heterogenous local area network. Chapter 2 introduces key concepts in the design of any virtual terminal service: terminal compatability, terminal drivers and terminal modules to serve the user requests. In particular, the design of the VAX/VMS Terminal Driver is discussed in some detail since our virtual terminal driver (or remote terminal driver) is based heavily on it.

Chapter 3 discusses the design and implementation of the virtual terminal service - called the Remote Terminal Access (RTA). The underlying network communication software (IPC) is introduced from the viewpoint of the IPC primitives that it offers to RTA. Then, the discussion goes into the design and implementation of the virtual terminal driver and the design of other terminal modules to support a dynamic client-server virtual terminal service. In an effort to formalize the protocols which RTA peer processes (at different nodes) must observe in order to communicate with each other, tables and descriptions of message sequences are provided, covering all the different message types and message sequences during the course of virtual network communication. The chapter ends with a discussion of flow-control strategies employed in this design.

Finally, chapter 4 summarizes the design and implementation by discussing the merits and shortcomings of this design. The chapter ends with suggestions for extensions that could be made to this implementation.

CHAPTER 2

OVERVIEW OF TERMINAL FUNCTIONALITY

Any effort in the design and implementation of remote terminal access requires an understanding of terminal compatability, terminal driver and terminal emulator functionality. These various topics are discussed in this chapter. In particular, the VMS terminal driver is explained is some detail since the remote terminal access implementation is modeled after the VMS driver.

2.1 Terminal Compatability

Video Terminals, or more commonly known as CRTs, come in a wide array of shapes, sizes and extended features. However, the common redeeming feature of most CRTs is that the code generated by depressing a particular key labelled, say "A", despite its various locations on different types of keyboards, is unique and follows the ASCII or EBCDIC code standard. See Fig. 2.1 [VT100]. Note that a 8-bit byte is enough to contain a unique representation of any particular ASCII code - character or otherwise. However, the range of

Octal Code	Char	Octal Code	Char	Octal Code	Char	Octal Code	Char
000	NUL	040	SP	100	@	140	`
001	SOH	041	!	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	045	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	'	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054	,	114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072	:	132	Z	172	z
033	ESC	073	;	133	[	173	{
034	FS	074	<	134	\	174	¦
035	GS	075	=	135	]	175	}
036	RS	076	>	136	^	176	~
037	US	077	?	137	—	177	DEL

Fig. 2.1 ASCII Code

byte values defined by the ASCII standard by itself was not enough for more sophisticated terminal applications - terminal screen operations, terminal graphics etc. To meet this need, terminals were made to support either DIGITAL's VT52 standard or the American National Standards Institute (ANSI) standard or both.

Essentially, the ANSI or VT52 standard is nothing new; both these standards were drawn from the ASCII standard. For example, a Cursor Up command in VT52 is represented by the ASCII codes ESC A. The ANSI standard, currently the newest version being ANSI X3.64, chooses to represent the same cursor command by ESC [A. Fig. 2.2 shows the ANSI/VT52 Cursor Control, Auxiliary Keypad Numeric Key and Auxiliary Keypad PF Key codes [GIGI]. In addition to these representations, ANSI provides flexible Mode Control Sequences which always starts off with the ASCII sequence ESC [and followed by a sequence of any one or more ASCII characters. These Mode Control Sequences provide the manufacturer with the flexibility in implementing a wide variety of terminal characteristics and options.

The bottom line, then, is that terminals connected directly to VAX/VMS systems must have ANSI/VT52 compatability since the VMS terminal input/output interface honors only ANSI/VT52 codes. Subsequently, terminals

Key	Keypad Numeric Mode	Keypad Application Mode ANSI	Keypad Application Mode VT52
0	0	ESC O p	ESC ? p
1	1	ESC O q	ESC ? q
2	2	ESC O r	ESC ? r
3	3	ESC O s	ESC ? s
4	4	ESC O t	ESC ? t
5	5	ESC O u	ESC ? u
6	6	ESC O v	ESC ? v
7	7	ESC O w	ESC ? w
8	8	ESC O x	ESC ? x
9	9	ESC O y	ESC ? y
—	—	ESC O m	ESC ? m
,	,	ESC O l	ESC ? l
.	.	ESC O n	ESC ? n
ENTER	Same as RETURN	ESC O M	ESC ? M

Auxiliary Keypad PF Key Codes

Cursor Key (Arrow)	VT52 Mode	ANSI Mode/Cursor Key Mode Reset	ANSI Mode/Cursor Key Mode Set (Application)
Up	ESC A	ESC [A	ESC O A
Down	ESC B	ESB [B	ESC O B
Right	ESC C	ESC [C	ESC O C
Left	ESC D	ESC [D	ESC O D

Cursor Control Key Codes

Key	Keypad Numeric Mode/ Keypad Application Mode ANSI	VT52
PF1/HARDCOPY	ESC O P	ESC ? P
PF2/LOCTR	ESC O Q	ESC ? Q
PF3/TEXT	ESC O R	ESC ? R
PF4/RESET	ESC O S	ESC ? S

Auxiliary Keypad Numeric Key Codes

Fig. 2.2 ANSI/VT52 Key Codes

connected to a remote system wishing to access the host VMS system must also be inherently ANSI/VT52 compatible or made so by software methods.

2.2 Terminal Drivers

The complexity of I/O devices and the need to share them in a multiprogramming environment precludes us from executing code that can directly manipulate I/O devices. In such a multiprogramming environment, the user interface to a peripheral device is provided by a defined set of system services. These system services, in turn, call up a set of kernel modules that access and control the device on behalf of the user. The complexities involved in synchronization with the rest of the system, such as buffered or direct I/O, recovery from device errors and timeouts, and other device-dependent operations, are taken care of by these kernel modules. These modules, collectively, are called the device driver for the particular physical device.

Viewed vertically, the implementation of the driver code as kernel modules subscribes to the layer strategy of multiprogramming systems. See Fig. 2.3 [VAX 84A]. The process, executing the driver code in kernel mode, has access to the complete set of instructions, registers and

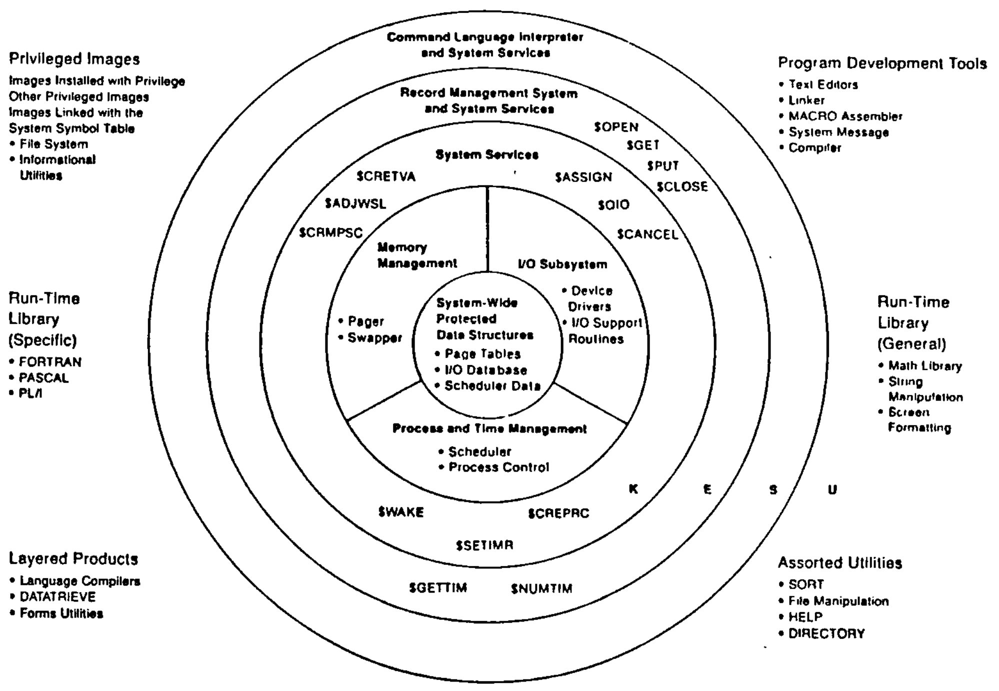

Fig. 2.3 Layered Design of the VAX/VMS Operating System

memory allowed in the kernel mode. Such access, although done in process context, is not directly accessible by the user layer. That is, the mode of the process determines its privilege for accessing memory and the types of instructions it can execute. Such a layered protection for different levels allows distinct functional areas to be constructed, one upon another, with each lower layer providing primitives for the next higher layer.

Viewed horizontally, a device driver is a set of routines and tables which are used by the system to process an I/O request for a particular device. In general, the operating system is primarily a collection of routines that the users call to perform various functions. Sharing of the processor by various processes is brought about by the process quantum expiration, or if a detected event causes a process of higher priority than the current process to become computable (via interrupts). In other words, the system is interrupt-driven.

Terminal Drivers, in particular, are kernel modules that make possible the interactive use of a particular system. It is useful to view the driver as having two interfaces: the external I/O interface between the outside world and the driver, and the internal I/O interface between the driver and the user process. To accommodate interactive use of the system, terminal drivers contain line discipline modules, which interpret input and output. In canonical mode, the line discipline converts raw data sequences typed at the keyboard (external interface) to a canonical form (what the user really meant) before sending the data to a receiving process (internal interface). The line discipline also converts raw output sequences written by a process to a format that the terminal screen expects. In raw mode, the line discipline passes data between processes and the terminal without such conversions [UNIX].

Typically, the external I/O interface of the terminal driver is provided by the terminal device, the terminal controller and a vector table containing the address of the interrupt-handler for that device. The function of the terminal controller is to arbitrate between the many terminal lines connected to it. By its hardware mechanism, each interrupting terminal device (unit) gets its chance to communicate with the system. The controller generates the interrupt to the system along with the identifying vector address of the interrupting unit. Using this address as the offset to the vector table, the system then accesses the interrupt servicing routine for the driver. For instance, the input interrupt-handler is activated when the user hits a key on the keyboard. The output interrupt-handler is activated when the screen is ready to accept the next byte and there is a byte to be sent.

On the other hand, the internal I/O interface of the driver is via system calls. These system calls are tailored to meet the needs of the specific terminal driver. Fig. 2.4 illustrates the internal and external I/O interface of the driver.

It is interesting to note that the first input character (usually a carriage return) from the external interface results in the driver spawning the login process. In turn, the login process performs internal I/O by prompting for the login name and reading the login name variable that is entered by the user. If the login sequence is correct, the login process spawns the shell process. From here on, there will always be a process (depending on what utility the user is summoning) which performs internal I/O to the driver. In short, the interactive nature of a terminal session is brought about, on one hand, by the user sitting in front of the terminal and, on the other hand, by a cooperating process performing internal I/O to the terminal driver.

2.2.1 Discussion the VAX/VMS Local Terminal Driver

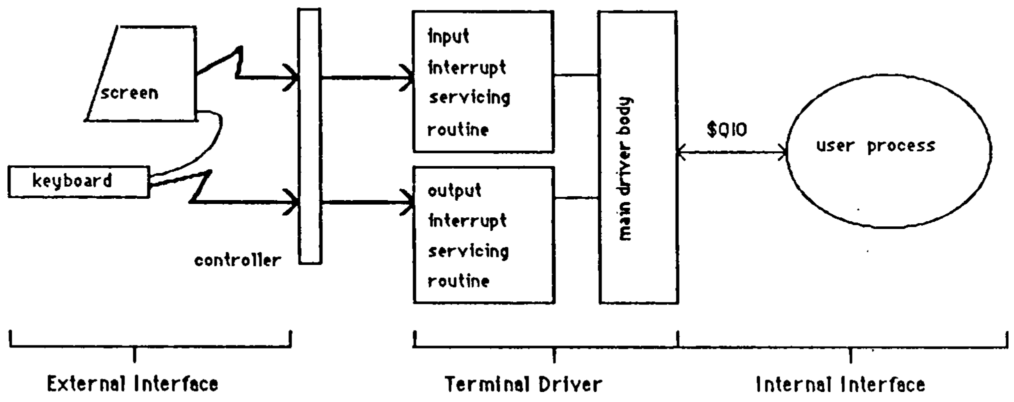

Fig. 2.4 External and Internal I/O Interfaces to a Terminal Driver

The features of the VMS Terminal Driver are discussed; most of these features should also be available in our Remote Terminal Driver. The main parts of the Terminal Driver code are then discussed, followed by an explanation of the pertinent I/O data structures that are managed by VMS during driver operation. After that, the synchronization techniques that are used during an I/O request are discussed, followed by a description of the sequence of events that are initiated by an external I/O (via interrupts) and internal I/O (via the $QIO system call). Finally, a login sequence is traced from the time when the user depresses the carriage return key on the terminal's keyboard to the initial entry into VMS by way of the command language DCL.

2.2.1.1 Features of the VAX/VMS Local Terminal Driver

The VAX/VMS Terminal Driver provides the following features and capabilities [VAXMAN A]:

* Input processing
  - Command line editing and recall
  - Control characters and special keys
  - Input character validation (read verify)
  - ANSI escape sequence detection

- Type ahead capability
- Specifiable or default input terminators
- Special operating modes, such as NOECHO and PASTHRU

* Output processing
  - Efficiency
  - Limited full-duplex operation
  - Formatted or unformatted output

* Dial-up support
  - Modem control
  - Hangup on logging out
  - Preservation of process across hangups

* Miscellaneous
  - Terminal/mailbox interaction
  - Autobaud detection
  - Out-of-band control character handling

All these features and capabilities are accessible to the user via a defined set of arguments to the $QIO system service. Table 1 [VAXMAN B] displays the set of arguments, called function codes and function modifiers, that are valid and meaningful only to the VMS Terminal Driver.

Terminal Driver

Functions	Arguments	Modifiers
IO$_READVBLK	P1 - buffer address	IO$M_NOECHO
IO$_READLBLK	P2 - buffer size	IO$M_CVTLOW
IO$_READPROMPT	P3 - timeout	IO$M_NOFILTR
	P4 - read terminator block address	IO$M_TIMED
		IO$M_PURGE
	P5 - prompt string buffer address	IO$M_DSABLMBX
		IO$M_TRMNOECHO
	P6 - prompt string buffer size[1]	IO$M_ESCAPE
IO$_READVBLK	P1 - buffer address	IO$M_EXTEND[2]
	P2 - buffer size	
	P3 - access mode to probe itemlist	
	P4 - (zero)	
	P5 - itemlist buffer address	
	P6 - itemlist buffer size	
IO$_WRITEVBLK	P1 - buffer address	IO$M_CANCTRLO
IO$_WRITELBLK	P2 - buffer size	IO$M_ENABLMBX
IO$_WRITEPBLK	P3 - (ignored)	IO$M_NOFORMAT
	P4 - carriage control specifier[3]	IO$M_REFRESH
		IO$M_BREAKTHRU
IO$_SETMODE	P1 - characteristics buffer address	
IO$_SETCHAR	P2 - characteristics buffer size	
	P3 - speed specifier	
	P4 - fill specifier	
	P5 - parity flags	
IO$_SETMODE	(none)	IO$M_HANGUP
IO$_SETCHAR		
IO$_SETMODE	P1 - buffer address	IO$M_BRDCST
	P2 - buffer size	
IO$_SETMODE	P1 - AST service routine address	IO$M_CTRLCAST
IO$_SETCHAR	P2 - AST parameter	IO$M_CTRLYAST
	P3 - access mode to deliver AST	
IO$_SETMODE	P1 - AST service routine address	IO$M_OUTBAND
IO$_SETCHAR	P2 - character mask address	IO$M_TT_ABORT[4]
	P3 - access mode to deliver AST	IO$M_INCLUDE[4]
IO$_SETMODE	P1 - address of control signals	IO$M_SET_MODEM[5]
IO$_SETCHAR		IO$M_MAINT
IO$_SETMODE	(none)	IO$M_LOOP[5]
IO$_SETCHAR		IO$M_UNLOOP[5]
		IO$M_MAINT
IO$_SENSEMODE	P1 - characteristics buffer address	IO$M_TYPEAHDCNT
IO$_SENSECHAR	P2 - characteristics buffer size	
IO$_SENSEMODE	P1 - address of input modem signal block	IO$M_RD_MODEM
IO$_SENSECHAR		
IO$_SENSEMODE	P1 - buffer address	IO$M_BRDCST
	P2 - buffer size	

Table 1 $QIO Function Codes and Modifiers for the VMS Terminal Driver

Furthermore, total VMS compatability extends to the:

* VMS Accounting - keeps tab on the resources, cpu time, etc.
* VMS Operator Communication Process (OPCOM) - allows any terminal defined by the user (usually a systems personnel) to be an operator's terminal. Also, any broadcast messages are sent via OPCOM to all terminal devices. Terminal sessions are recorded in the Operator Communications log file.
* VMS Error Logger subsystem - any errors in the driver are recorded in the error log file which can be read by the SYE Utility.

At this juncture, it will suffice to say that the device driver must be written under VMS conventions and must be loaded by the VMS Utility in a specified way so that the device becomes integrated into the VMS operating system. Details on how to install a VMS driver are described in [VAXMAN H].

2.2.1.2 Main Components of the Driver Code

Under the VAX/VMS operating system, the Terminal Driver is, as are all VMS device drivers, a set of routines and tables that the system uses to process an I/O request for a particular device type. Before the $QIO routine actually accesses the specific driver, it performs a standard set of functions and checks common to all devices; for example, validating those arguments of the I/O request that are not device specific, allocating system buffers for the I/O request etc. Such preprocessing is called VMS I/O preprocessing [VAXMAN C]. After the preprocessing, $QIO calls the specific driver; the driver gains control and device-specific processing commences. When the driver has completed all the device-specific tasks (according to the specified $QIO arguments), control is returned to the $QIO. Subsequently, $QIO does some common postprocessing functions, called VMS I/O postprocessing, such as returning allocated buffers to system memory, and other housekeeping operations. Finally, $QIO returns control to the user process that originated the system call [VAXMAN C]. The following is a description of the main parts of the terminal device driver code that play a key role in carrying out the device-specific processing [VAXMAN D]. See Fig. 2.5.

* Function Decision Routines (FDT Routines)

The VMS Terminal Driver has many FDT routines to

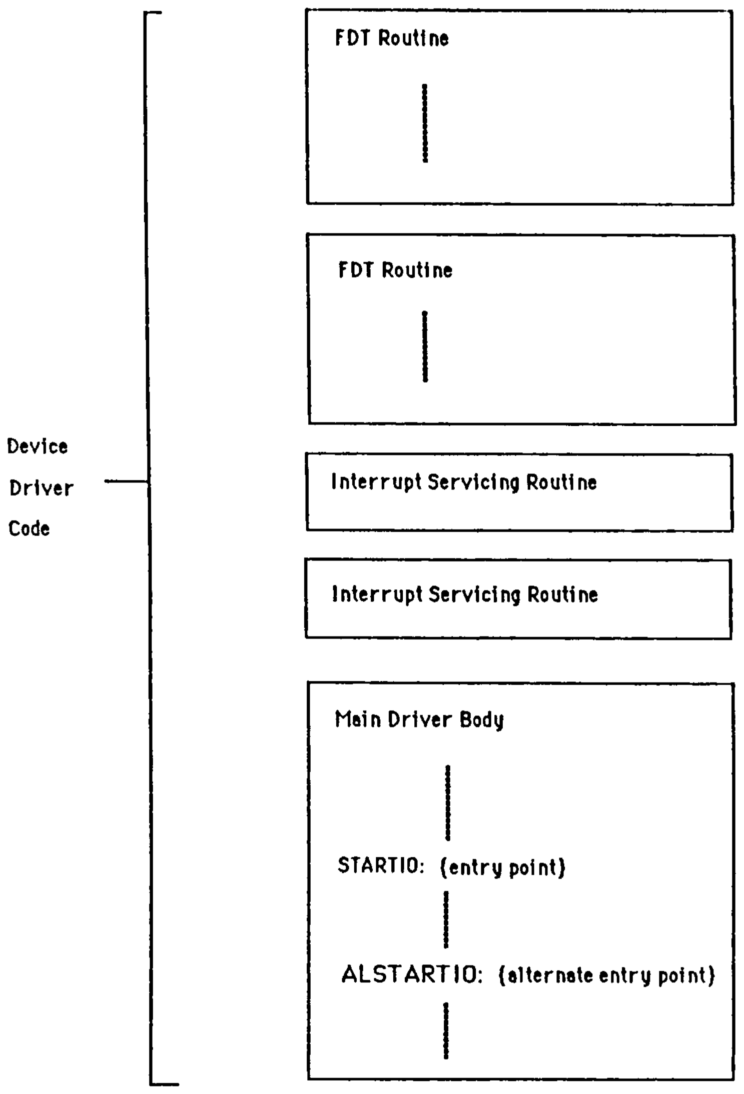
FDT Routine
FDT Routine
Device
Driver
Code
Interrupt Servicing Routine
Interrupt Servicing Routine
Main Driver Body
STARTIO: (entry point)
ALSTARTIO: (alternate entry point)

Fig. 2.5 Main Parts of Device Driver Code

preprocess $QIO requests. Table 1 gives an idea of the number of FDT routines that are in the Terminal Driver. For example, the $QIO Read request with a prompt parameter allows the user to request a read operation from the terminal, while at the same time outputting a string to the terminal. The FDT Read routine will pack the output string with the read buffer in a specified way and call the main driver routine. In short, FDT routines accept a specific $QIO request and interpret the variable parameters for that request and pack the interpreted information at a specified block to be passed to the main driver routine.

* Main Driver Routine (STARTIO)
  This routine analyzes the I/O function and branches to the driver code that prepares specific device for that I/O operation. There are many more functions that the main driver routine performs on behalf of the $QIO request. In short, the FDT interprets the $QIO device-specific request, and the main routine performs according to the specifications of the request.

* Alternate Driver Routine (ALSTARTIO)
  This routine is entered from some FDT routines. The main difference between this entry point and STARTIO is one of

synchronization concern. It will be discussed later.

* Interrupt Servicing Routines
  A hardware originated device interrupt will ultimately trigger the correct interrupt servicing routine if the device driver follows the VMS driver conventions. In the terminal driver, there are two interrupt servicing routines, one for reading input from the keyboard and one for outputting a character to the CRT.

2.2.1.3 Driver Data structures

The VMS I/O database is a collection of data structures that provide various information to the VAX/VMS operating system and drivers to help monitor the status of, and control the functions of, the I/O subsystem. The following is a description of the major data structures that pertain to the terminal driver operation [VAXMAN E].

* Unit Control Block (UCB)
  Each device unit on the system has its own UCB. The UCB describes the device type, its current status, current I/O activity. It also provides pointers to other data structures. The UCB, within this discourse, can be viewed as the central data structure from which other data

structures and crucial device status information can be obtained. All UCBs belonging to the same controller type are linked together. A UCB field indicates whether the device is busy (i.e., a fork process is currently using the device).

* Device-Data Block (DDB)
  Represents the generic device name and driver name for a set of devices attached to the same type of controller. For example, the DZDRIVER controller for all TT terminal devices share one DDB. That DDB points to the first UCB (i.e., TTA0:) Also, DDBs are connected by a linked-list.

* Driver-Dispatch Table (DDT)
  This data structure points to the entry points of the driver code e.g. FDT routines, STARTIO and ALSTARTIO entry points. Each DDB points to a common DDT for the same controller.

* Channel-Request Block (CRB)
  The activity of each controller is described in the CRB. It also contains pointers to the driver's Interrupt-servicing routines and to the corresponding IDB (see below).

* Interrupt-Dispatch Block (IDB)

The IDB records is corresponding controller characteristics. It also contains the memory-mapped address of the Central Status Register (CSR). The CSR, one for every terminal controller, is the only register the interrupt-handlers of the driver need to access and control the specific terminal unit. The terminal controller takes care of the arbitration and multiplexing responsibilities for its designated terminal units.

* I/O-Request Packet (IRP)

When a user queues a valid I/O request by issuing a $QIO or $QIOW system service, the service creates an IRP. The IRP contains a description of the request and receives the status of the I/O processing as it proceeds. The IRP is queued to the UCB when the UCB indicates that a fork process is currently executing the device unit. See Section 2.3 for its relevance to the monitor concept in the main driver code.

* System Control Block (SCB)

The SCB is a table containing the vectors used to dispatch (software and hardware) interrupts and exceptions. An external interrupt will result in the hardware switching the system to an interrupt context, followed by the accessing of one of these vectors to transfer control to.

Fig. 2.6 describes the data structures which are traversed by VMS when the external I/O occurs via interrupts and when internal I/O occurs via the $QIO system call. It is important to note here that despite the different set of data structures that are traversed , the objective of VMS is always to track down the correct UCB of the interrupting terminal device, the correct CSR address of the interrupting device's controller, and the correct driver code for the device - in this case the VMS Terminal Driver.

2.2.1.4 Synchronization of I/O processing

Synchronization of I/O processing involves three main synchronization techniques [VAX 84B]:

(1) Interrupt Priority Levels (IPL)

In an interrupt-driven system like VMS follows a convention of Interrupt Priority Levels (IPL). There are 32 levels of IPLs :

* IPL 0 - User-mode software
* IPL 1 - 15 - Software interrupts

  Initiated by the MACRO SOFTINT.

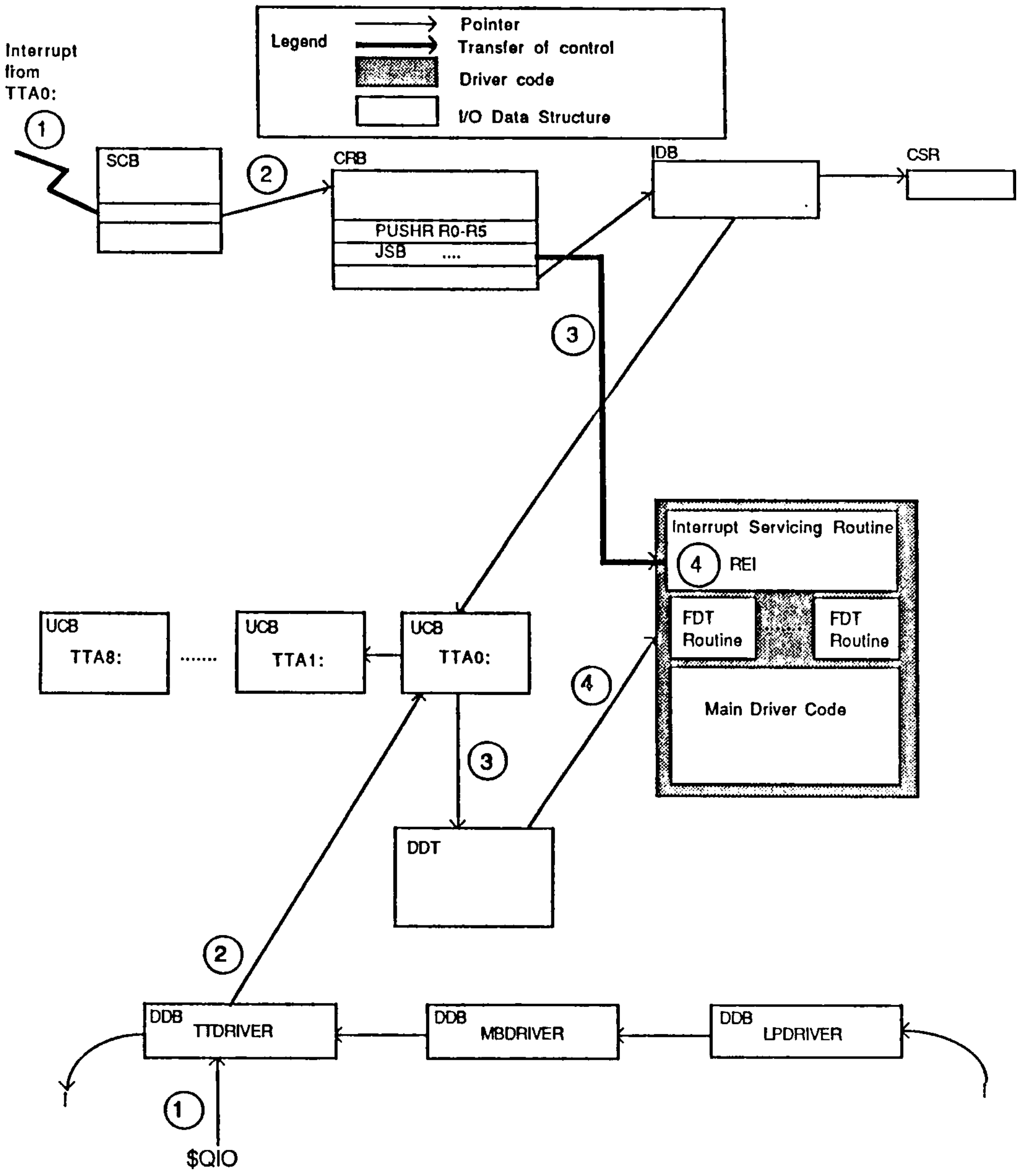

Fig. 2.6 I/O Data Structure traversals by VMS

* IPL 16 - 31 - Hardware interrupts
  Initiated by external interrupts.

Because higher IPLs take precedence over lower IPLs, a routine executing at one IPL can block interrupts at the selected level and all lower IPLs. The VMS operating system assigns progressively higher IPLs to progressively important events. For example, the following is a descending order in priorities: machine check (IPL 31), device interrupts (IPL 20-23), device driver processes (called fork processes IPL 8), AST delivery (IPL 2) and the user-mode process (IPL 0).

(2) Fork Processing

Fork Processing is the technique that allows device drivers to lower IPL in a manner consistent with the interrupt nesting scheme defined by the VAX architecture. When a device driver receives control in response to a device interrupt, it performs whatever steps are necessary to service the interrupt at device IPL. For example, any device registers whose contents would be destroyed by another interrupt must be read before the driver dismisses the device interrupt. Usually, however, there is some processing that can be deferred; to execute at device IPL for extended periods of time would slow down the system. The driver signals that it wishes to delay further processing until the IPL in the system drops below a predetermined value, the fork IPL associated with the driver. This signaling is accomplished by calling system routine G^EXE$FORK that saves the address of the next instruction in the driver in a data structure called a fork block. The fork block is then inserted at the end of the fork queue for that IPL value (8 through 11) and a software interrupt at the appropriate IPL is requested.

(3) Synchronization within the Driver

Synchronization within a device driver, in particular, requires the implementation of the "monitor" concept. For example, when the FDT routine has completed all the device-dependent preprocessing at IPL 2, it calls the system routine via JSB G^EXE$QIODRVPKT (a MACRO instruction) to execute the main driver code at STARTIO at IPL 8. That is, it is an event waiting to enter the monitor. Now, if there was a previous incomplete write operation executing the driver code (i.e., in the monitor) that was prempted by some higher IPL, entering the driver code just because the processor grants the process access at IPL 8 would erroneously overwrite the write buffer. Thus, to avoid this, the system routine will check to see if the code is being executed by a process (a fork process). If not, the system creates a fork process to execute the code starting at STARTIO. Otherwise, the system routine queues the request (represented by the I/O Request Packet) in a queue and removes the next IRP in the queue to execute in fork process context. Thus, the entry at STARTIO is the entry to a monitor. Only one fork process can execute at a time in the same device unit. The queue waiting on entry is priority based (according to the caller's base priority) and FIFO. Fig.2.7A describes the implementation of the monitor concept.

There are some instances when it is necessary to bypass the monitor implementation in the main driver code. That is, a process, on gaining the CPU, may want to enter the main driver code as long as it is at IPL 8. It does not care whether there is actually another process waiting to enter nor whether the same device unit is actually being executed by another fork process. Bypassing the monitor constraint is done by calling G^EXE$ALTQUEPKT in the FDT routine. Fig. 2.7B describes the execution of ALSTARTIO - the alternate startio entry point to the main driver code.

This feature is used in the VMS local terminal driver when a user process does a $QIO Write system call to the terminal driver and the terminal line characteristic ALTYPEAHEAD is set. The driver will process the Write IRP even though there may be a Read IRP waiting to enter the driver.

This feature will also be used in the implementation of the remote terminal driver, albeit for a different purpose.

2.2.1.5 The Sequence of Events during Internal and External I/O.

This section describes the sequence of events which are initiated by a $QIO Write request and another sequence of events which are initiated by an input interrupt. It is important to emphasize that movement from one event to another is by either creating fork processes or by

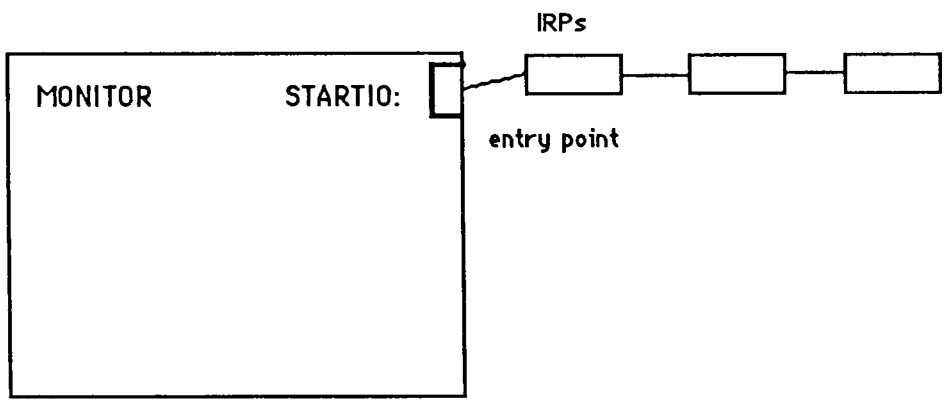

Fig. 2.7A Monitor Construct at IPL 8

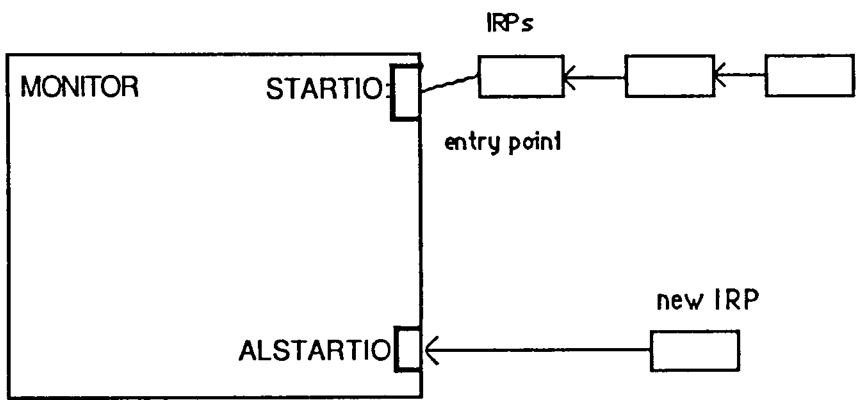

Fig. 2.7B Bypassing the Monitor Construct

directly raising the IPL level.

Fig. 2.8 describes the sequence of events which occur during a $QIO Write.

(1) The user issues a $QIO to a terminal device to write a string of characters. The user process executes in process context in user mode.

(2) The $QIO system service gains control in process context but in kernel mode. It performs device-independent processing of the I/O request e.g. validating channel number, checking that the process does not exceed process' quotas, etc.

(3) The $QIO system service uses the driver's function decision table to decide which function decision routine within the driver code it will execute.

------------------- DRIVER CODE --------------------

(4) The appropriate function decision routine TTY$FDTWRITE in the driver code is executed in process context in kernel mode. In this routine, the

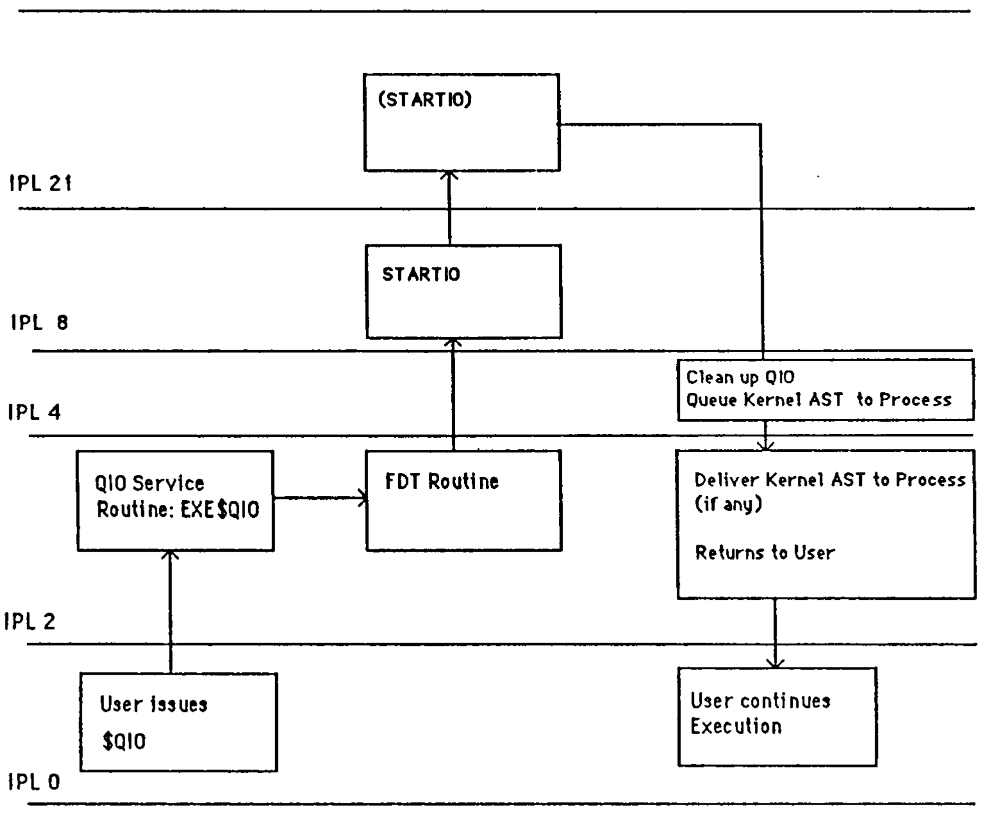

Fig. 2.8 $QIO Write Events

user buffer that contains the string is checked to see if it can be accessed. Also $QIO buffered I/O operations with the specific parameters to a terminal driver device are interpreted. Thus, this routine (and all FDT routines) performs device-specific tasks for the $QIO. Finally, a system buffer is constructed containing the character string and a wealth of information concerning specific $QIO write buffered operations. This system buffer, along with other information, is passed to the main driver code called STARTIO. All FDT preprocessing routines execute in full process context but in kernel mode.

(5) STARTIO - this is the main driver code and executes in fork process context (explained later). Its first statement is a CASE statement to decide which FDT routine it originated from. In this case, it branches to the location DO_WRITE. At DO_WRITE, the current cursor position, and other perfunctory duties of the terminal driver are performed and any additional control characters are inserted into the output buffer. This output buffer thus consists of:

  * the original character string
  * control characters prefixing and suffixing the character string as requested by $QIO arguments.
  * control characters that are inserted for cursor control and other perfunctory duties of the terminal driver.

The system buffer resides in the terminal driver waiting to be output. On a output-ready device interrupt from an UART, the driver's interrupt servicing routine, executing in interrupt stack, temporarily "freezes" all other activity and outputs contents of the buffer byte by byte to the CRT. The driver returns to $QIO for device independent post-processing

-----------exiting driver code---------------

* $QIO post processing cleans up the $QIO request e.g. copying the status of the iosb block, etc. It queues a special Kernel Ast back to the original user process. It has to do this because the user process may not be active any more.

* VMS returns control to the user process

* User process executes the next instruction

Fig. 2.9 describes the events that occur when initiated by an input interrupt. From this figure, it is clear that external interrupts have much higher priority over the rest of the system.

Terminal I/O (input via the keyboard, an output on the CRT) are two separate interrupt-driven events. For the sake of simplicity, we will concentrate on movement of a byte of data to and from the Central Status Register (CSR). VAX microcode and controller hardware take care of the movement of the data to and from to the correct terminal device via the UNIBUS.

2.2.1.6 The Logging Process

The following is an attempt to explain the sequence of events that the user goes through to log into the VMS system. The sequences are explained graphically in Fig 2.10A, Fig. 2.10B, and Fig. 2.10C.

2.3 Remote Terminal Modules

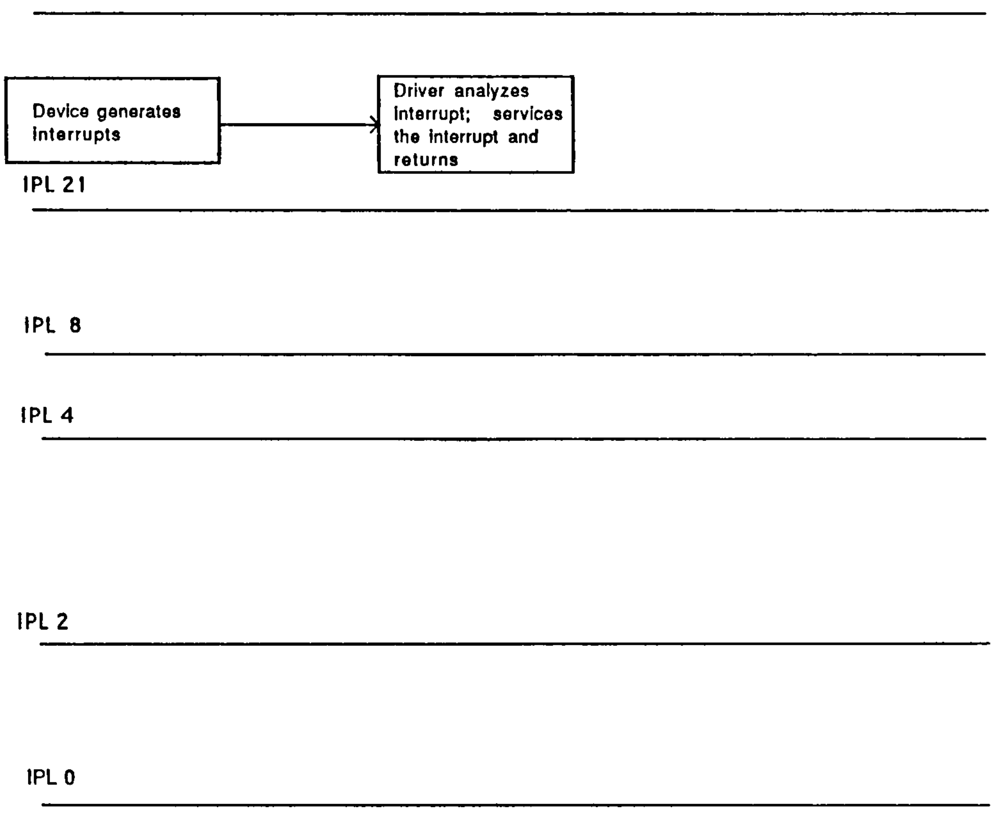
Device generates interrupts
Driver analyzes interrupt; services the interrupt and returns
IPL 21
IPL 8
IPL 4
IPL 2
IPL 0

Fig. 2.9 Interrupt initiated Events

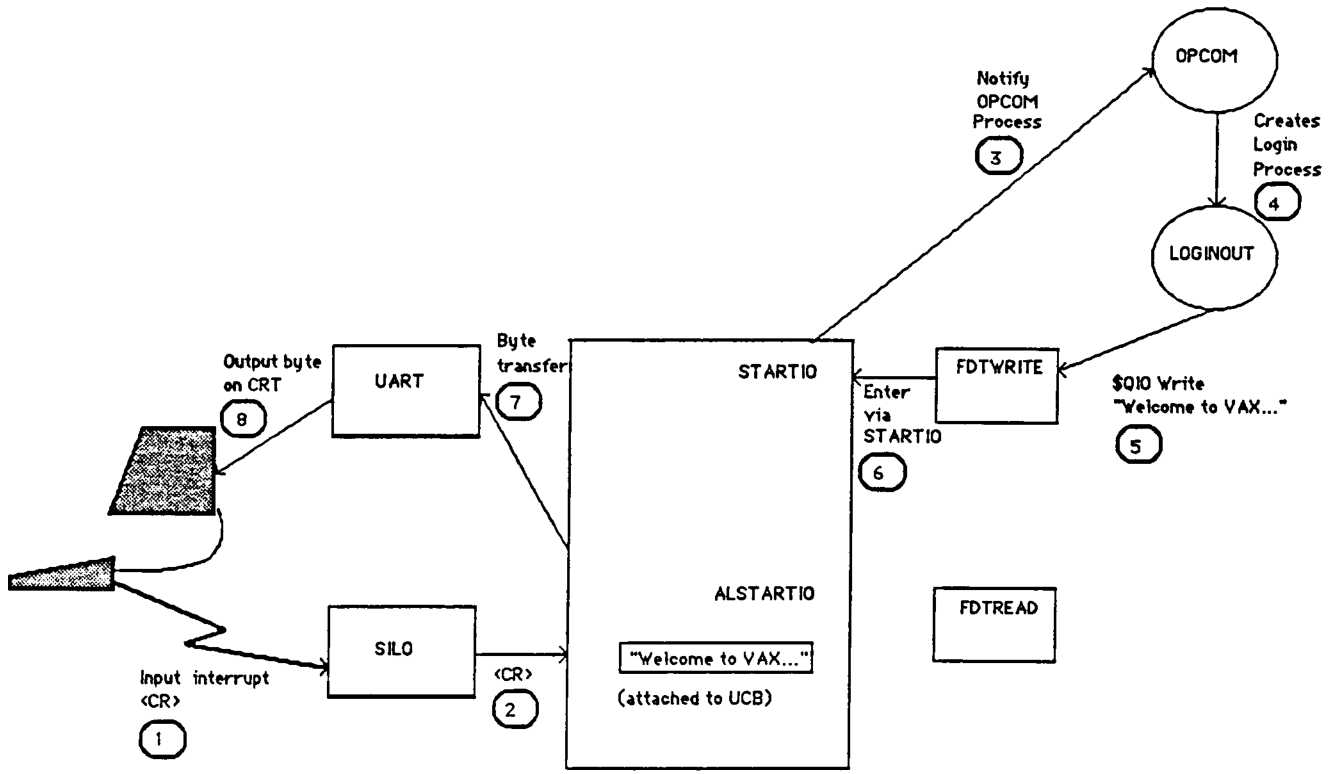

Fig. 2.10A Schematic Diagram of User Logging into a VMS System via the VMS Terminal Driver

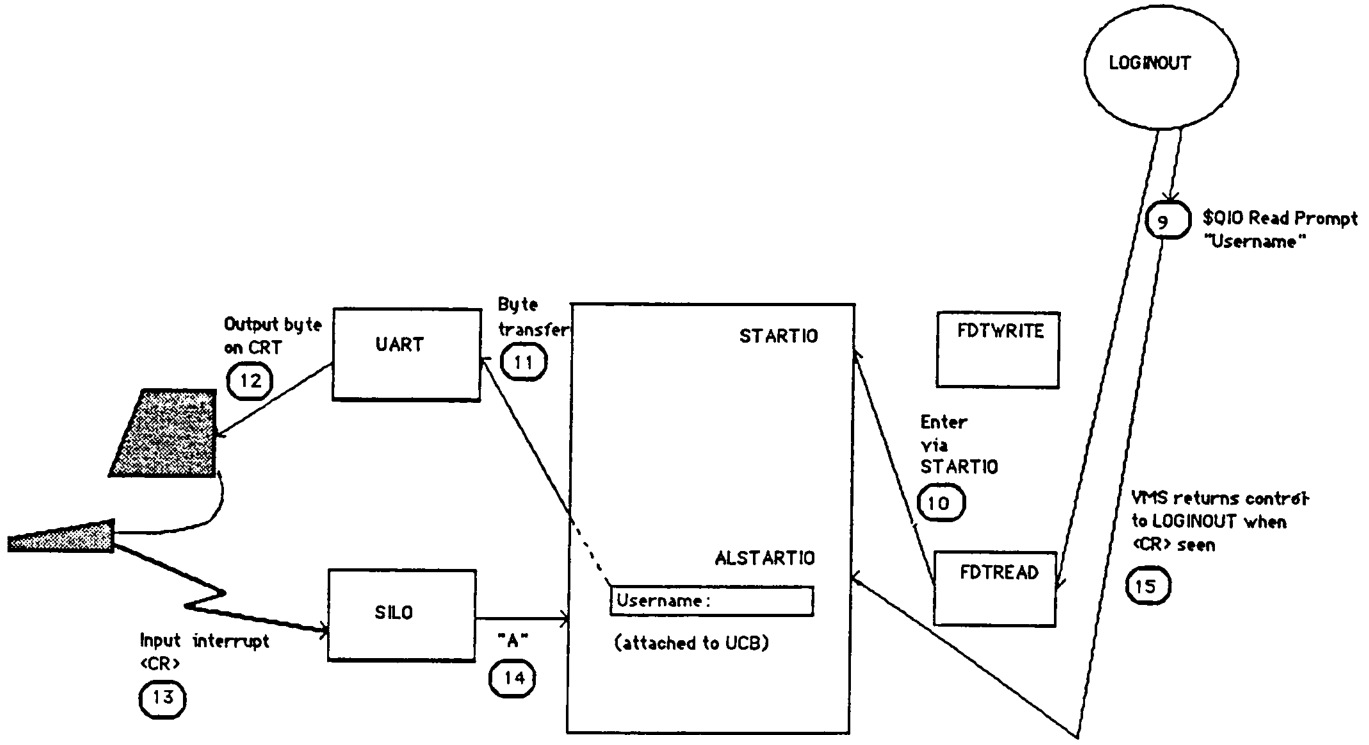

Fig. 2.10B Schematic Diagram of User Logging into a VMS System via the VMS Terminal Driver

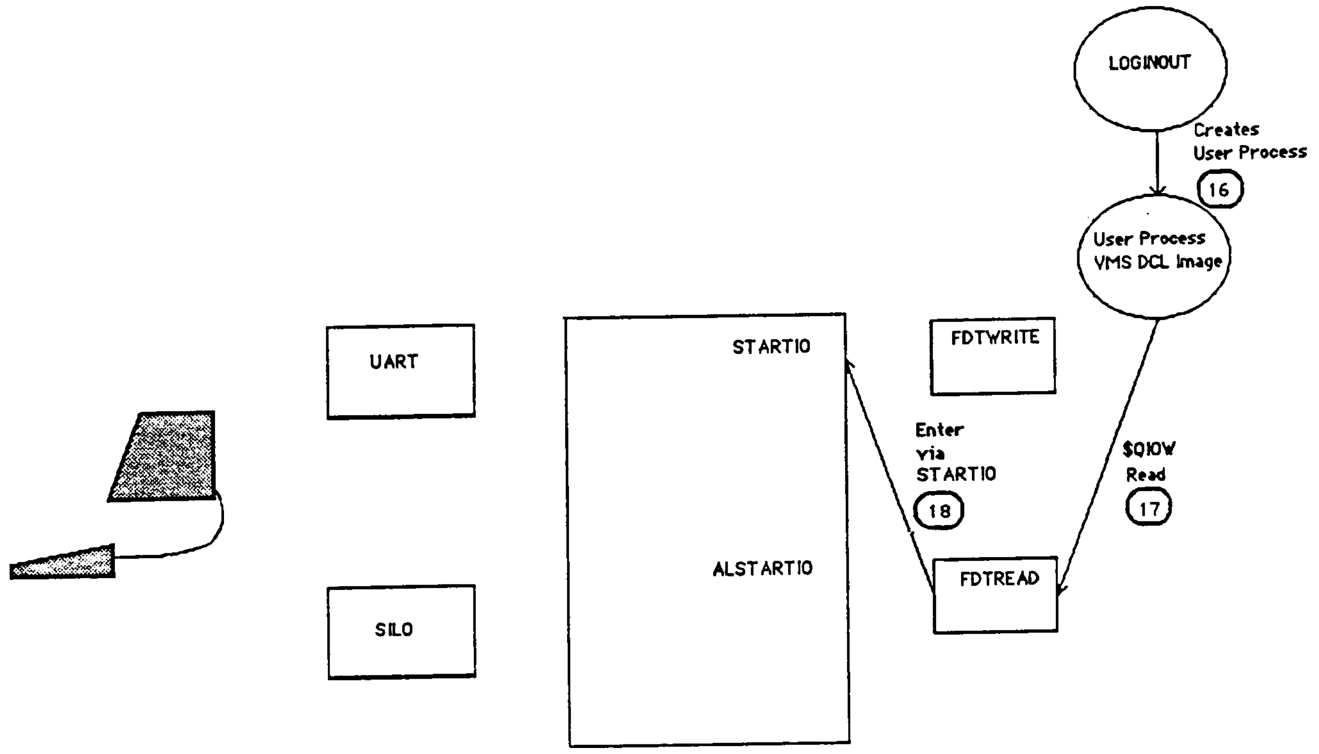

Fig. 2.10C Schematic Diagram of User Logging into a VMS System via the VMS Terminal Driver

There are many instances when one autonomous computer system (the remote node) needs to remotely access another autonomous computer system (the host node). Despite the different physical mediums used to connect these systems, be it high-speed coaxial cables or low speed asynchronous lines, there are two remote host software modules that are common in all implementations seeking to remotely access the host node from a remote node. These modules are:

* Terminal Emulator

Terminal emulators are software modules that are located at the remote node and provide virtual terminal service to the host node. That is, a user at the remote node can remotely access the host node, as if his remote terminal were directly connected to the host node.

The functions of the terminal emulator are:

(1) to pass input from the remote terminal's keyboard to the host node in its raw mode.

(2) to pass output from the host node to remote terminal's screen in its raw mode.

(3) to perform key code mapping if the remote terminal is not ANSI/VT52 compatible.

(4) to perform key relocation if the remote terminal does not have the same key locations as the standard host terminal. For example, DIGITAL's VT100 compatible terminals have a special keypad for editing functions. A terminal emulator on a remote node wishing to remotely access a DIGITAL host node, where the remote terminal does not have this keypad, will map the keypad locations on other available keys on the keyboard.

* Terminal Servers

Terminal servers are background processes running at both the remote and host nodes that set up the virtual terminal connection between the remote terminal and the host. In addition, housekeeping functions like keeping a maximum on the lines that can remotely access the host system, and recovery from line errors are the responsibilities of the servers.

CHAPTER 3

DESIGNING AND IMPLEMENTING THE REMOTE TERMINAL ACCESS

The design of Remote Terminal Access (RTA) adheres to the layered concept of the Open Services Interconnection (OSI) Reference Model of the International Standard Organization (ISO). As seen in Fig. 3.1, the RTA encompasses the Session and Application layer of the OSI model. We call this the RTA Layer.

The peer processes, which constitute the RTA layer, utilize the primitives provided by the underlying Inter-Process Communication (IPC) layer for virtual communication with other peer processes on other nodes. Such virtual communication between peer processes follow rules and conventions specific to that layer- known collectively as the Virtual Terminal Protocol.

The discussion on the design of RTA will begin with a brief description of the IPC primitives that are used by the RTA layer for virtual communication. The discussion then moves on to the design of RTA; it is is conveniently divided into three sections: the design of the Remote Terminal Driver, the Terminal Emulator and the Terminal Servers. The

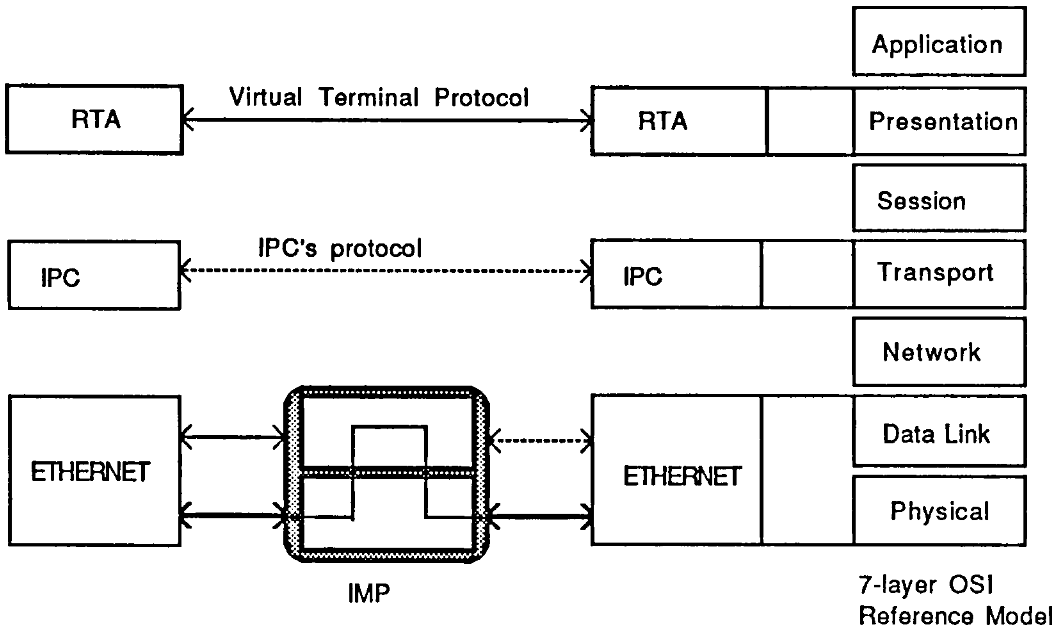

Fig. 3.1 RTA layers with respect to the OSI Reference Model

next section encapsulates the design of RTA by formalizing the Virtual Terminal Protocol. The approach taken here is based on DECNET's formalization of its Network Terminal Protocol [VAXMAN G]. Finally, the last section discusses the end-to-end flow control strategies that are used in the RTA.

3.1 The Inter-Process Communication

The interprocess communication protocols are in the heart of any distributed/networking system [HIL 86]. They are responsible for exchanging data reliably between processes within the same machine and, more importantly, between different machines. The IPC interface to the RTA layer consists of primitives in three categories: naming and addressing, datagram and virtual circuit services [HIL 86]. The following is a brief description of these three categories of the IPC. For a detailed description of the IPC, the reader should refer to [IPC].

3.1.1 Naming and Addressing Primitives

In order for a process to communicate with a remote process, the process must acquire a socket which identifies the process address at the particular node. Since there are only a fixed number of sockets available, the sockets are to be returned to the IPC after usage. IPC provides primitives to get local and remote sockets, and to return them.

The primitives are:

(1) Getsocket:

This primitive requests a socket for a process from a local IPC. To get a socket, the process must supply the PUN. When the request is honored, a socket number is returned to the requesting process.

(2) ReturnSocket:

This primitive will release a socket back to the local IPC when it is no longer needed. Processes that die will have their sockets automatically returned to the local IPC.

(3) FindRemoteSocket:

This primitive will search for a remote socket, given the PUN as the input parameter.

3.1.2 Datagram Services

IPC provides a reliable datagram service to the RTA. That is, when a process (a peer process in RTA) at one node uses this service, IPC will make sure that the datagram will arrive at the destination node.

The primitives are:

(1) SendDG:

This primitive will reliably send a variable-length datagram to the remote process. The send parameters include a remote and local socket, and a buffer which contains the message to be sent. A maximum length is imposed on the message.

(2) RcvDG:

This primitive will reliably receive variable-length datagrams. The parameters include a local socket, a buffer pointer, and a buffer length.

3.1.3 Virtual-Circuit Services

The IPC provides the RTA with the perfect channel where there are no errors and all packets are delivered in order. In other words, a virtual circuit simulates a physical connection between two communicating peer processes.

The primitives are:

(1) OpenVC:

This primitive establishes a VC between two processes. The input parameters include the local socket, the remote socket, and an optional window size n (in order for the RTA process to exchange VC initiation packets). The "open" returns a VC number if the two processes agree to open a circuit and returns an error if the process on the other end has rejected the proposal.

(2) Listen:

This primitive is a kind of passive open because it does not result in any packets being sent or any connections being established. A local process uses this primitive to inform the local IPC of its willingness to open a VC if a remote process attempts to activate a connection. This mechanism speeds up establishing a VC by recording an entry for it in the VC table with incomplete information. The "Listen" parameters include a local socket and a remote host. The "Listen" returns a VC number of a one-side passive VC.

(3) SendVC:

This primitive sends a variable-length packet on a specific VC to the remote process. The parameters of this primitive include a VC number and a message (message pointer and length). A maximum length is imposed on messages.

(4) RcvVC:

This primitive receives a variable-length packet on a specific VC from a remote process. Packets are only accepted in sequence. The parameters include a VC number, a buffer pointer, and a buffer length.

3.1.4 Examples of Virtual Communication on a Virtual Circuit

The order in which the different IPC primitives are actually used depends on the objectives of the RTA peer processes using them. Fig. 3.2 provides an example of how two RTA peer processes, A and B, at separate nodes set a virtual circuit over time t. Process A will then send a packet through its Virtual Circuit to B.

3.2 Design of a Remote Terminal Driver

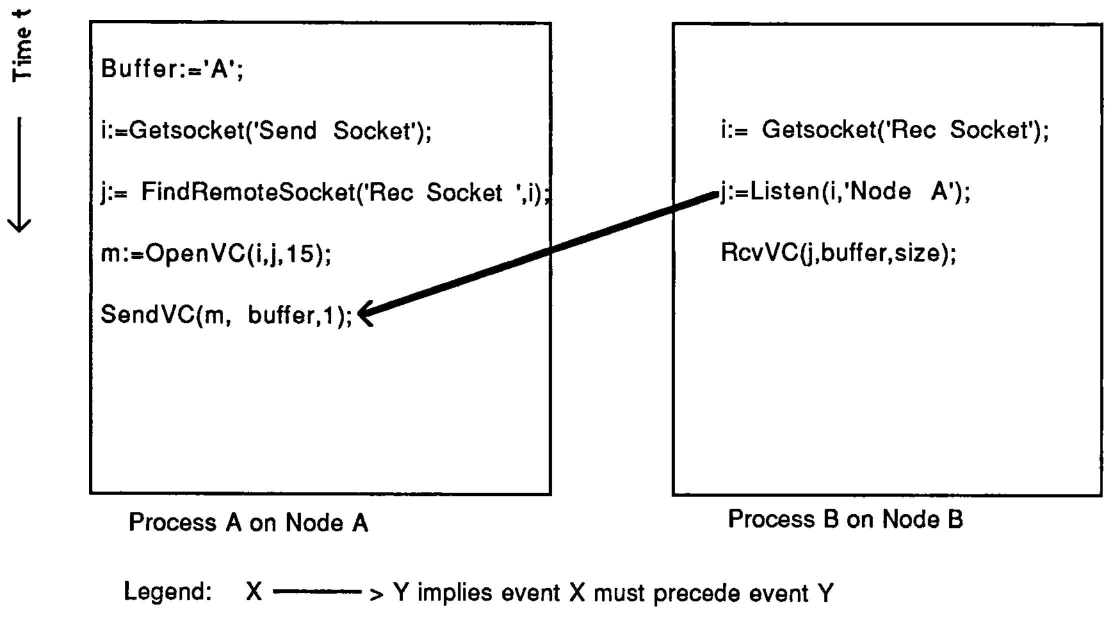
Time t
Buffer:='A';
i:=Getsocket('Send Socket');
j:= FindRemoteSocket('Rec Socket ',i);
m:=OpenVC(i,j,15);
SendVC(m, buffer,1);
i:= Getsocket('Rec Socket');
j:=Listen(i,'Node A');
RcvVC(j,buffer,size);
Process A on Node A
Process B on Node B
Legend: X ———— > Y implies event X must precede event Y

Fig. 3.2 Example of how two RTA peer processes A and B establishes a Virtual Circuit to send data from A to B

The internal interface (via $QIO system calls) to the Remote Terminal Driver must be identical to that of a VMS Terminal Driver. Only then would all VMS utility programs and all user programs which use the system call interface to the VMS Terminal Driver also work for the Remote Driver.

As for the external interface to the driver, the problem was twofold. First, a method had to be established by which an ordinary user process could communicate with the driver. This was because the user process, acting as an intermediary, also communicated with the rest of the network system. Secondly, this method had to simulate external interrupts from the terminal device.

A tempting solution may be to somehow use the existing VMS Local Terminal Driver to provide remote terminal access. The internal interface via the $QIO system call is readily available. As for the external interface, we would have to somehow simulate hardware interrupts from the terminal keyboard and screen so that the external I/O interface to the driver originates not from the directly connected asynchronous terminal device but from the remote terminal. Since VMS allows privileged user programs to initiate software interrupts via the MACRO instruction SETIPL, there may be a way to initiate software interrupts from the intermediary process which would, in turn, activate the interrupt-handlers of the VMS Driver. Unfortunately, the IPL range for software interrupts is only from IPL 1 through 15. Consequently, the possible range of interrupt-handlers that are addressed as offsets to the SCB vector table does not include the interrupt-handler of the terminal driver at IPL 21 - only accessible by a hardware interrupt.

Another alternative may be to somehow use DECNET's remote terminal driver. Again, the internal interface via $QIO is readily available. However, as Fig. 3.3 [VAX 84C] illustrates, we have to know the format of the internal IRP and the proper address from which to insert and remove "internal IRPs" (they form the external interface to the driver). Such knowledge requires an intricate knowledge of DECNET's Remote Terminal Driver design - which we have not.

Given that we cannot use any of the available VMS terminal drivers, we are left with the task of developing one. The $QIO system service, being the standard internal interface to all VMS drivers, was chosen as the internal interface to the driver. Since the internal interface must meet all VMS Terminal Specifications, the required programming (in VMS MACRO with calls to system routines) was substantial. This task, while being a tedious one, was not insurmountable.

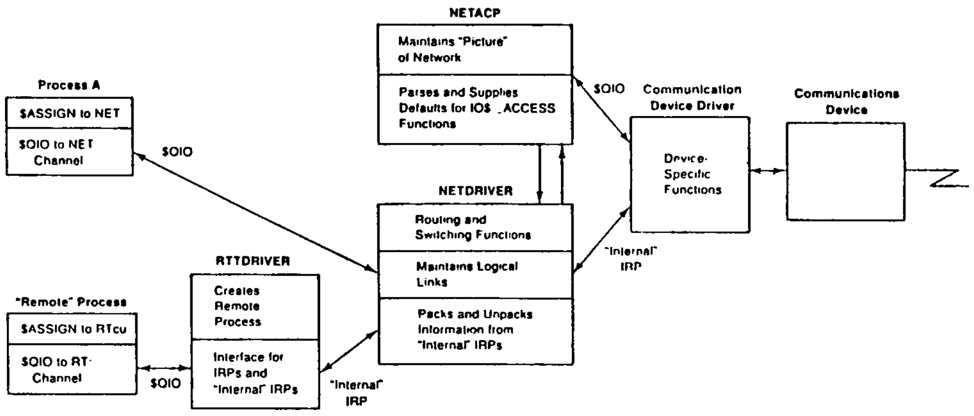
NETACP
Maintains "Picture" of Network
Parses and Supplies Defaults for IO$_ACCESS Functions
Process A
$ASSIGN to NET
$QIO to NET Channel
$QIO
$QIO
Communication Device Driver
Device-Specific Functions
Communications Device
NETDRIVER
Routing and Switching Functions
Maintains Logical Links
Packs and Unpacks Information from "Internal" IRPs
"Internal" IRP
RTTDRIVER
Creates Remote Process
Interface for IRPs and "Internal" IRPs
"Remote" Process
$ASSIGN to RTcu
$QIO to RT-Channel
$QIO
"Internal" IRP

Fig. 3.3 Internal IRPs as the External Interface to the VMS Terminal Driver

To provide the external interface to the driver, the $QIO system service was also chosen as the way to communicate with the driver. Since this mode of entry is normally the internal interface's way of communicating with the driver, special function modifiers are selected to distinguish these simulated interrupt requests from regular $QIO request. Henceforth, simulated $QIOs will be referred to as Network $QIO as opposed to regular $QIOs. Table 2 shows the valid function codes and function modifiers that are understood by the Remote Terminal Driver (recall Table 1 for valid codes of the VMS Terminal Driver).

Given that $QIO system service as the means of communicating with the driver, the design of simulating interrupts was approached by looking at the Input and Output Interrupts separately.

3.2.1 Output-Ready Interrupts (UART)

To recap on the events that occur when a process request a regular $QIO Write to a VAX/VMS Driver, we will begin by emphasizing that, in the VMS driver, the regular $QIO Write Requests operate asynchronously with Output-Ready Interrupts. A process issuing a $QIO Write request would write a string into a specified system buffer, the

Remote Terminal Driver

Functions	Arguments	Modifiers
:		
:		
:		
{ same as Table 1 - for the VMS Terminal Driver }		
IO$_READVBLK	P1 - BUFFER ADDRESS P2 - SIZE	IO$M_NETWORK IO$M_EXTEND
IO$_WRITEVBLK	P1 - BUFFER ADDRESS P2 - SIZE	IO$M_NETWORK IO$M_EXTEND

Table 2 $QIO Function Codes and modifiers for the Remote Terminal Driver

Output-Interrupt line is enabled and control would be returned to the user. When the screen of the CRT is ready to accept another byte, the interrupt-servicing routine for the output to the CRT is activated, and byte transfer to the CRT occurs transparently to the calling process. Fig. 3.4A illustrates this sequence of events.

Simulating the Output-Ready Interrupt means that whenever a user process performs a regular $QIO Write request entering the driver code via STARTIO, the buffer contents of the accompanying regular IRP must be "written" to the remote CRT before the control passes back to the user process requesting the $QIO Write. In this implementation, the contents of the $QIO Write IRP's buffer are copied to the buffer of a pending $QIO Network Read IRP - see Fig. 3.4B. The process that performed the pending read IRP, called here the UART process, must have previously issued a Network $QIO request.

Fig. 3.4C traces the IPL transition due to events described in Fig. 3.4B. The crucial features used in synchronization of different events are the usage of IPLs and fork-processing. The VMS post-processing of the Network IRP is queued at IPL 4, and followed by the queuing of regular Write IRP. When this happens, VMS actually post-processes the Network IRP first, followed then by the

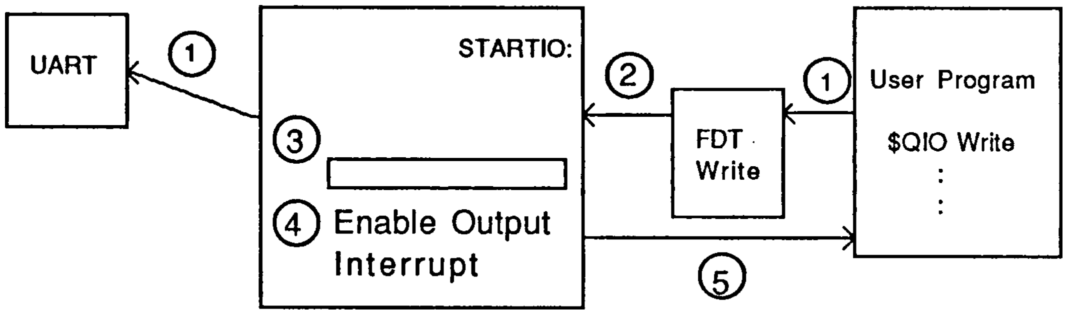

Fig. 3.4A Output-Ready Interrupts in a VMS Terminal Driver.

Sequence:

ASYNCHRONOUS EVENTS

1 User Program calls $QIO Write.
2 FDT Write does preprocessing, and enters Driver via STARTIO.
3 The driver puts string into a system buffer (pointed to by the IRP of the $QIO Write request).
4 The driver enables the Output-Interrupt for the line.
5 The driver returns control to User process.

1 Output-Ready interrupt activates the UART (when the line is free and Output-Interrupt for the line is enabled. The contents of the system buffer are discharged a byte at a time.

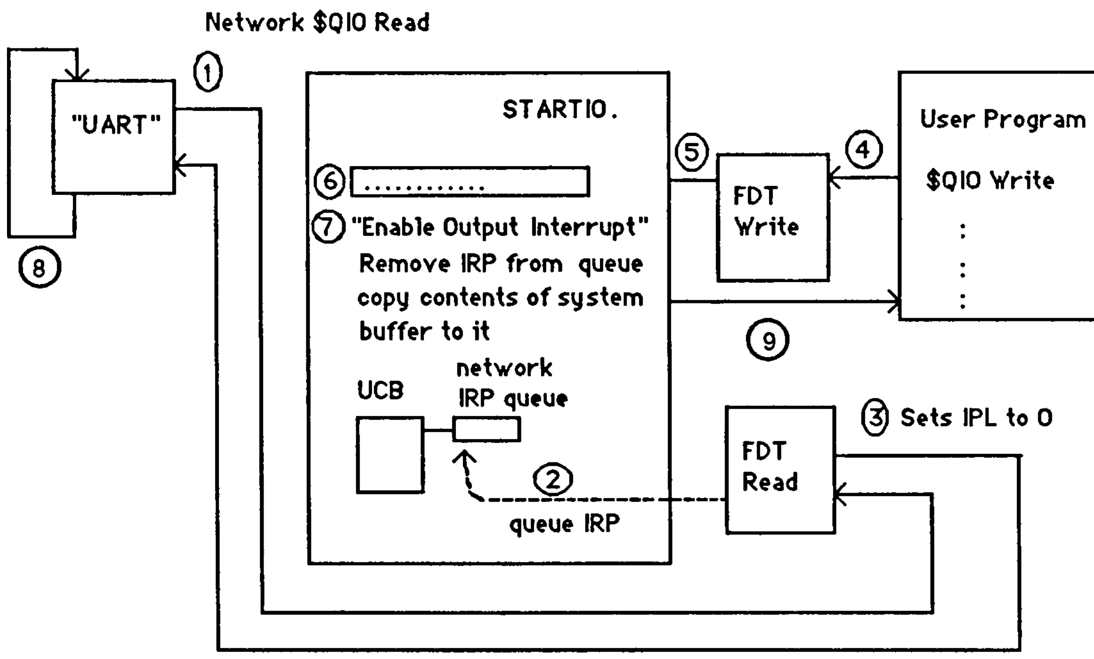

Fig. 3.4B Simulating Output-Ready Interrupts in a Remote Terminal Driver

1 "UART" process does a Network $QIO Read.
2 FDT Read processes the special $QIO Read request by queing the IRP of the Network Read Request into a network IRP queue.
3 FDT Read routine calls a special VMS routine to set the IPL back to 0, and make the UART process wait.
4 User process does a regular $QIO Write.
5 FDT Write enters the main driver code via STARTIO.
6 The driver puts string into a system buffer (pointed to by the IRP of the $QIO Write request).
7 "Enables Output Interrupt" for the line by removing the Network Read IRP from the network IRP queue and transferring the contents of its system buffer into the Network IRP's system buffer space. The Network IRP is then queued for postprocessing at IPL 4.
8 VMS removes the Network Read IRP from its FIFO queue at IPL 4 and queues again at IPL 2 for special kernel AST processing. The AST is required to return control back to the UART process. Next IRP in the FIFO queue at IPL 4 is the regular write IRP. Again, postprocessing is done at IPL 4 and it is queued at IPL 2 for special kernel ast to return control to the User process. Notice that the UART process is always one step ahead of the User process in IPL queuing. Hence, the special kernel AST will return control first to the UART process, which in turn, queues a Network $QIO Read request again after it has discharged the read string of bytes through the network virtual line (via IPC). After all this only will VMS return control to the User Process.
9 The Driver initiates the return of control to the user by queuing the regular IRP for postprocessing at IPL 4.

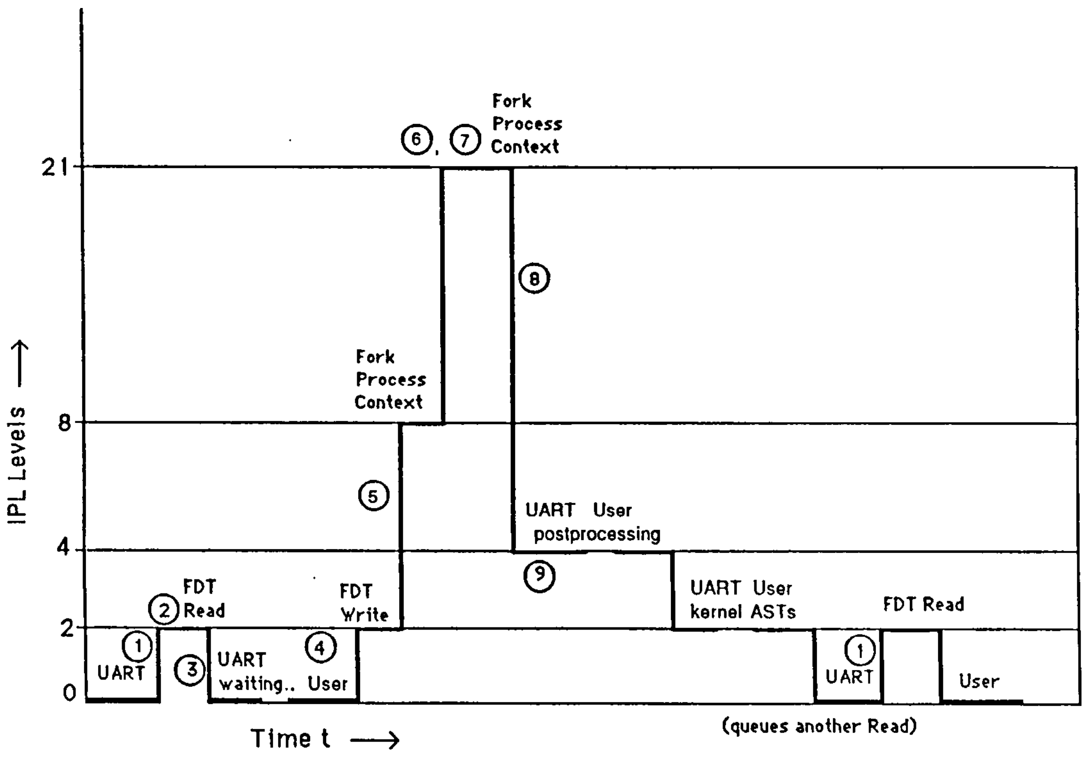

Fig. 3.4C IPL transitions for events described in Fig. 3.4B

regular IRP. After post-processing, VMS queues a special kernel AST at IPL 2, again, for the Network IRP first, followed by the regular IRP.

The UART process gains the processor first at IPL 0. This process now has the output string intended for the CRT. In our implementation, the process uses a network primitive to transfer the string to the remote process. The network primitives performs many system calls to finally transport the string to the remote process. After that, the UART process issues another Network $QIO Read. When the UART process returns from the driver FDT, pending its $QIO Read completion, the VMS Scheduler finally returns control to the user process.

From preceeding discussion, it may seem that the VMS Scheduler always returns control to the UART process before returning control to the user process. However, this is not the case because the VMS Scheduler may reschedule processes due to resource wait or process quantum expiration. Therefore, depending on the system activity, the user process may "overtake" the UART process. In that case, the driver does not attempt to copy the contents of the $QIO Write IRP's buffer into the buffer of the pending Network $QIO Read - simply because there is no pending Network $QIO Read IRP yet. Instead, the driver will append the contents of the $QIO Write buffer to the last non-empty space of a storage buffer. This buffer, part of the UCB buffer space, starts at UCB$L_NETREAD_STORE. The size of the buffer, at UCB$W_NETREAD_CT, is incremented accordingly and control is returned to the user process. Then, when the UART process is able to perform another Network $QIO Read, all the contents in the UCB buffer space are transferred to the buffer of the Network Read IRP and the UCB$W_NETREAD_CT is equated to zero. Control is then returned to the UART Process.

It should be noted that there may be many occurrences of the "overtake" condition resulting in an accumulation of bytes in the storage buffer. The question, then, is how large should the storage buffer be? In an empirical fashion, we arrived at the size of 800 bytes, citing that there was not a single instance of the accumulation of bytes exceeding the buffer's size at 500 bytes in our many testings under different loads. In any case, code has been included in the driver that would merely return control to the user process when this happens. Essentially, the output that was intended to appear on the screen would then be lost, as remote as the case may be when such a situation would arise.

A much greater inadequacy is the inability of the TERM_CALLEE process to delete the UART process. In an ideal situation, TERM_CALLEE should be able to delete UART and SILO processes after the user has logged off the remote terminal. While there are no complications involved in deleting a SILO process, deleting an UART process leads it to a miscellaneous RWAST state because VMS is unable to properly release UART's channel to the remote terminal unit. At the present time, the author does not possess the full knowledge of those restricted VMS routines required to complete the deletion of the UART process. Thus, for the present, the problem of deletion of UART is circumvented by moving the inter-process communication, originally in the UART, to another process called UART_COMBINE. UART would still retain the remote terminal drivers Network $QIO Read interface and UART_COMBINE and UART would communicate with each other through a common mailbox. Then, when the user has logged off the remote terminal, TERM_CALLEE will delete the SILO and UART_COMBINE processes. Fig. 3.4D illustrates the scheme.

3.2.2 Input Interrupts (SILO)

Input Interrupts are simulated in another way. Unlike the

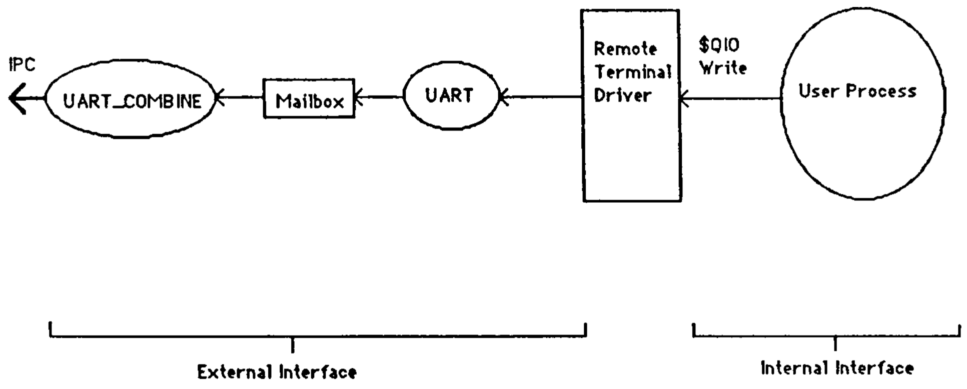

Fig. 3.4D Scheme of processes to simulate Output-Ready Interrupts

regular $QIO Write requests, the regular $QIO Read requests are obviously dependent on the Input Interrupt to provide the input. Fig. 3.5A illustrates the sequence in which the VMS Terminal Driver's input interrupt-servicing routine store the input byte in the type-ahead buffer. If there was a pending regular $QIO Read request, the byte is transferred to the regular IRP's buffer. If the terminal characteristic in a UCB field for the unit had the echo characteristic set, the byte is moved to an output buffer and the line is Output-Enabled. The interrupt servicing routine duly terminates and the pending regular $QIO read request is satisfied. The Input-interrupt Servicing routine processes one byte at a time because the echo byte is full-duplex (with no local CRT echo) and because many VMS applications have different kinds of responses depending on the type of byte input.

Simulating the Input Interrupt means that whenever there is an input from the remote keyboard, the byte must be transferred eventually to the type-ahead buffer. In this implementation, the process which receives the byte from the remote keyboard and does the Network $QIO Write to the driver is called the SILO process. When the SILO process does the abovementioned, and if there is a pending regular $QIO Read request, the driver moves the byte to the regular $QIO Read's IRP and the pending User Process is completed.

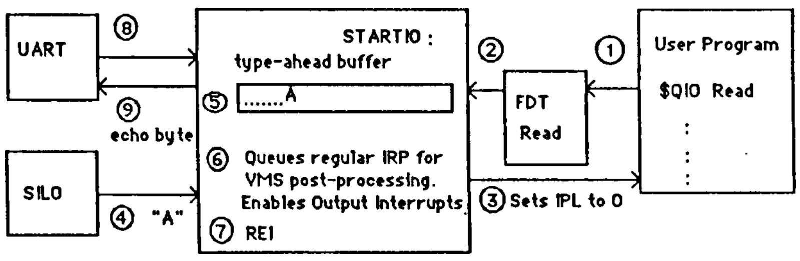

Fig. 3.5A Input Interrupts In a VMS Terminal Driver

1 User Program calls $QIO Read

2 FDT Read does preprocessing, enters driver via STARTIO

3 The type-ahead buffer is empty, so Driver calls special VMS Routine to set IPL back to 0.

4 Input interrupt activates the SILO when a key, say "A" is depressed at the keyboard. The character "A"
is passed to the interrupt-servicing routine.

5 The driver puts the byte into the type-ahead buffer (pointed to by UCB of the terminal unit).

6 If Read request is pending (as is the case), SILO first moves the byte from the type-ahead buffer into the regular Read IRP's system buffer and then queues the regular, pending Read IRP for VMS postprocessing at IPL 4. The driver then enables Output-Interrupt for full-duplex echo.

7 The driver returns from interrupt by the REI instruction.

8 Output-Ready interrupt activates the UART (when UART is free and line is enabled)

9 The driver outputs a byte to UART (if any). In most cases the byte is the echo character (full duplex mode) and returns from interrupt by the REI instruction.

10 Eventually, VMS postprocessing at IPL 4 gets its turn at the processor, and queues a special kernel AST at IPL 2 to return control to the user process. Later, at IPL 2, the special kernel AST queues return control to the process.

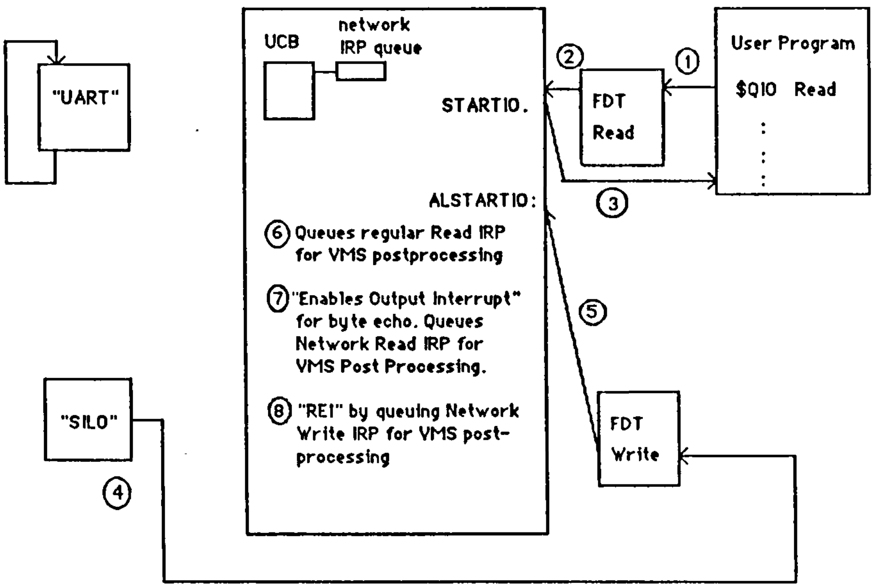
"UART"
UCB
network IRP queue
STARTIO.
ALSTARTIO:
⑥ Queues regular Read IRP for VMS postprocessing
⑦ "Enables Output Interrupt" for byte echo. Queues Network Read IRP for VMS Post Processing.
⑧ "REI" by queuing Network Write IRP for VMS post-processing
②
①
User Program
$QIO Read
:
:
FDT Read
③
⑤
"SILO"
④
FDT Write

Fig. 3.5B Simulating Input interrupts in a Remote Terminal Driver

Precondition: steps 1,2 and 3 are already done. Refer to Fig. 3.5A.

1 User program calls $QIO Read.

2 FDT does preprocessing, and enters the main driver code via STARTIO.

3 The type-ahead buffer is empty, so Driver calls a special VMS routine to set the IPL back to 0.

4 "SILO" process calls Network $QIO when it receives a character "A" via the virtual line.

5 FDT Write processes the Network $QIO request by moving the byte "A" into a system buffer pointed to by its IRP. Then, the FDT routine enters the main driver code via ALSTARTIO.

6 The driver moves the byte "A" into the system buffer of the pending, regular write IRP's system buffer and queues the VMS postprocessing for the UART process at IPL 4.

7 "Enables Output Interrupt" by moving byte "A" into the pending Network Read IRP's system buffer and queues the VMS postprocessing for the UART process at IPL 4. This, in effect, "echoes" the input byte.

8 The driver returns control to the "SILO" process by the REI instruction. This will result in the queuing of the VMS postprocessing at IPL 4.

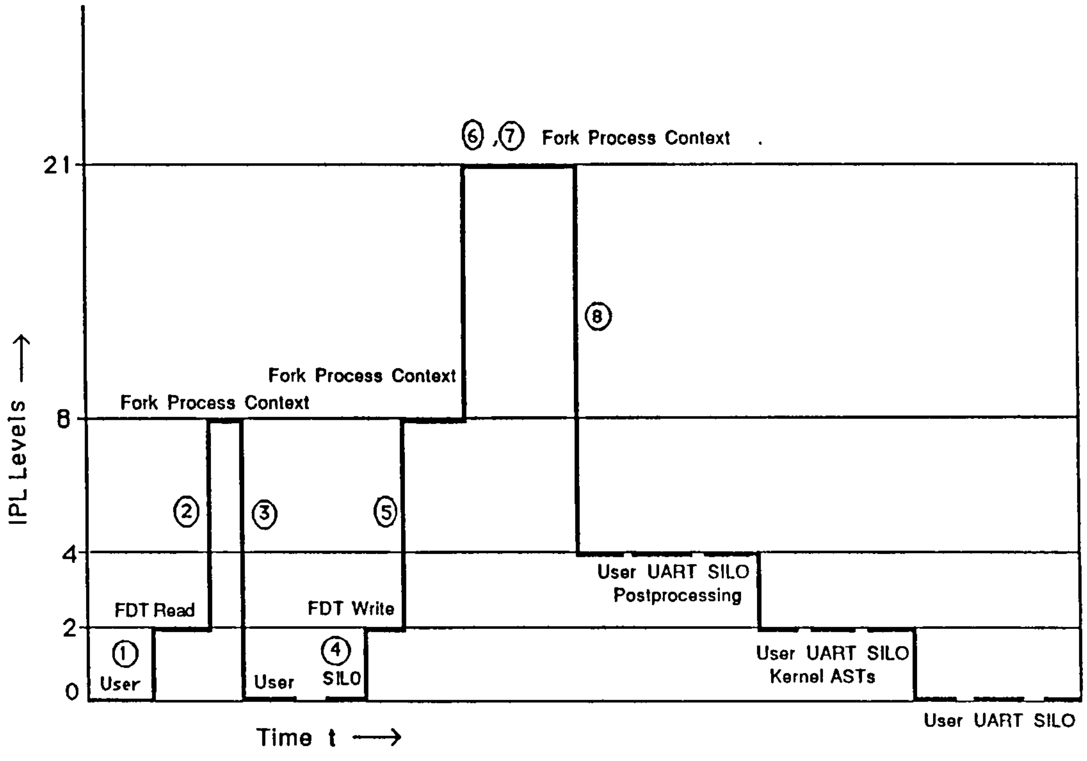

Fig. 3.5C IPL transitions for events described in Fig. 3.5B

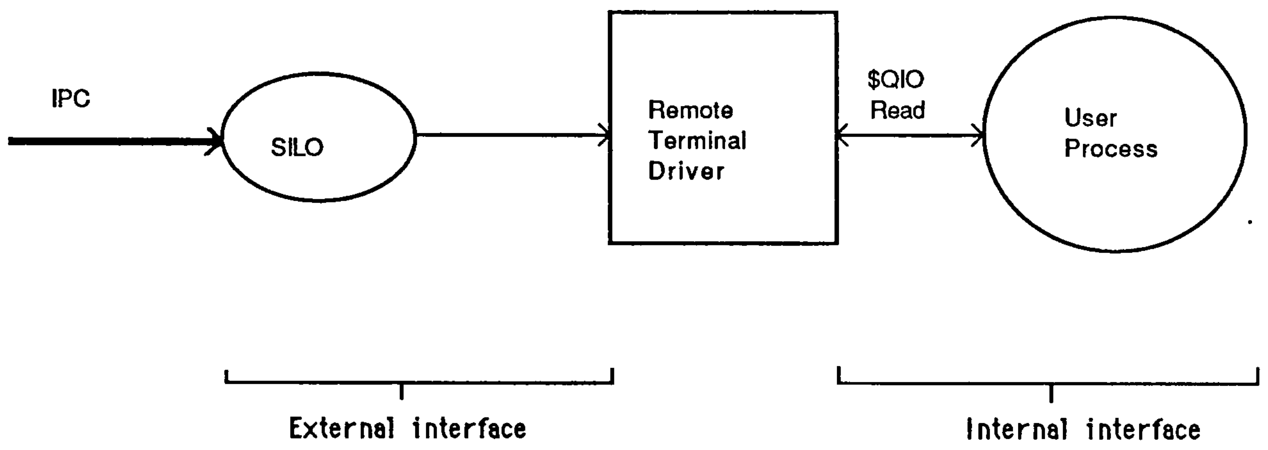

Fig. 3.5D Scheme of Processes to Simulate Input Interrupts

Fig. 3.5B illustrates this. Since the echoing of input bytes requires the feature of a pending Network $QIO Read IRP (see Section 3.2.1), the problem of "overtaking" is again empirically solved by appending the contents of the $QIO Write IRP's buffer to the last non-empty space in the storage buffer. Fig. 3.5C traces the IPL transitions due to the events described in Fig. 3.5B. The crucial features used to synchronize events here are:

(1) Usage of IPLs

(2) Fork Processing

(3) The entry point of the SILO process at ALSTARTIO.

This entry bypasses the monitor constraint at STARTIO where many pending regular $QIO Read requests may be waiting to enter the monitor (driver code)

Finally, Fig. 3.5D illustrates the design as a scheme of processes (recall its counterpart in Fig. 3.4D).

3.2.3 Special Driver UCB fields

The implementation of SILO and UART processes require the addition of new fields and the manipulation of an existing field in the UCB data structure.They are:

(1) UCB$Q_SAVE_STATE (new field)

The existing UCB field - UCB$Q_TT_STATE - reflects the current state of the Terminal Driver. For example, the state may be a READ state, with sub-states EDITOR, CONTROLO etc. All in all, there are more than 100 states that reflect exactly the current state of the driver. Now, as shown in Fig. 3.5C, the user process eventually returns to IPL 0. This allows the UART and SILO processes to reissue the Network $QIO requests. The problem here is that VMS I/O preprocessing (before FDT device-dependent processing) erases the UCB$Q_TT_STATE field. Since the SILO and UART processes are supposed to simulate interrupts, all the fields in the UCB should not change during "interrupt servicing". To meet this requirement, all changes to UCB$Q_TT_STATE made within the driver code (numerous locations) were also made to UCB$Q_SAVE_STATE. Then, when a SILO or UART process enters the FDT, the UCB$Q_TT_STATE (which had been erroneously cleared) is restored by the field UCB$Q_SAVE_STATE.

(2) UCB$W_WREFC (new field)

Every $ASSIGN system service call to a particular device unit will increment the existing UCB$W_REFC field corresponding to the device unit by one. Since both SILO and UART processes are separate processes requiring individual channels to the same device unit, the UCB for the corresponding device unit will have its UCB$W_REFC field incremented by two. Now, since both these processes are supposed to simulate interrupts, the count should not be incremented. To meet this requirement, the FDT Write routine decrements the UCB$W_REFC count by two at the initial entry of SILO process's first Network $QIO Write request. The field UCB$W_WREFC, acting as a flag, is then set to indicate that the decrement should only be done once. All following Network $QIO Write requests should not decrement the UCB$W_REFC field; after all, only the first Network $QIO Write request (by the SILO) should decrement UCB$W_REFC by two to nullify the preceding two $ASSIGNs by the SILO and UART processes - which incremented UCB$W_REFC by one each.

(3) UCB$L_IT_RIRPFL and UCB$L_IT_IRPBL (new field)
These fields are used to queue the Network IRPs as described in Section 3.2.1

(4) UCB$L_PID (old field)
This field must be cleared after the UART and SILO process have assigned a channel to the terminal unit. If this field is not cleared, no other user process will be able to assign a channel to the same terminal

unit except those processes that have the same username (or login name).

In summary, the SILO, UART and UART_COMBINE processes, residing in the host node, provide the external interface to the driver. It should be noted here that, so far, the details of network inter-process communication aspects have been avoided. For example, the preceding discussion only mentions that the UART process receives a byte from the driver and writes it to a "remote CRT". The discussion of inter-process communication will be deferred to the later section.

3.3 Design of the Terminal Emulator

From the discussion of the previous section, it is apparent that the SILO and UART_COMBINE processes residing in the host node are to have complementary or partner processes at the remote node. The partner processes residing at the remote node, in effect, function as the Terminal Emulator.

The process, TERM_READ, at the remote node, is the communicating partner of SILO process at the host node. TERM_READ simply reads byte by byte from the remote terminal's keyboard and passes the byte, via the IPC virtual circuit service, to the SILO process. Similarly, TERM_WRITE at the remote node, receives a string of bytes from UART_COMBINE via the IPC virtual circuit service and writes the string of bytes to the remote terminal. Fig. 3.6 is a schematic representation of TERM_READ and TERM_WRITE processes at the remote node and their complementary processes at the host node. The actual code of these processes is illustrated by Fig. 3.7. As can be seen, a total of two virtual circuits are required for one virtual terminal line.

The task of setting up virtual lines, amongst other functions, is the responsibility of the Terminal Server Process on both the remote and host node.

3.4 Design of the Terminal Server

In our implementation, the functions of the Terminal server are:

(1) to set up virtual terminal lines by request of the user

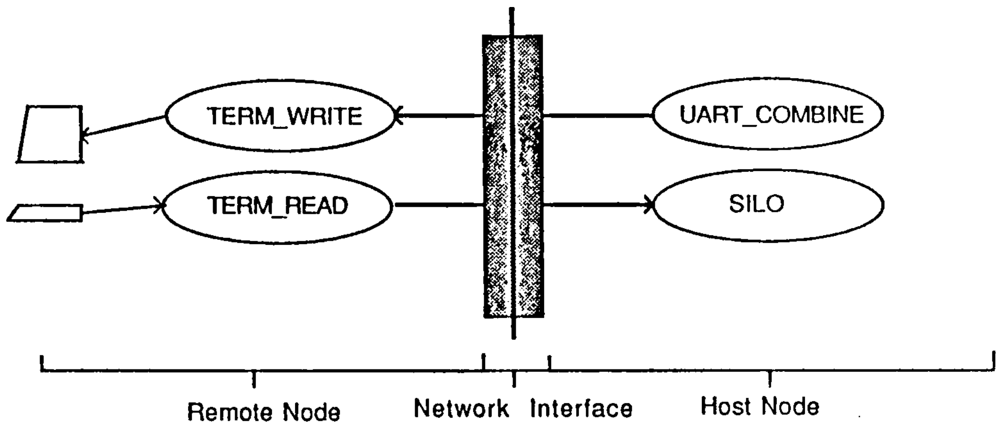

Fig. 3.6 Scheme of Processes for a Virtual Terminal Line.

```
program term_write(input, output);
    :
begin
    i:= Getsocket('TW    CSV3'); { get a local socket number }
    j:= Listen(i,CSV3 );            { Listen at remote node for remote socket}
    while forever do
     begin
      RcvVC(j,Buff, size);          {read data from the VC}
     {write to the terminal}
     state:=$QIOW(func:=io$_writevblk, chan:= device_channel1, p1 := Buff,
   end;               p2:=size);
end.
```

```
program uart(input,output);
    :
begin
    :
  i:= Getsocket('UART    CSV3');   { get a local socket }
 { get a remote socket number}

  j:=Findremotesocket('TW    CSV3',i);
  m:=OpenVC(i,j,15);
  while forever do
   begin
    status:= $QIOW( func:= io$_readvblk, chan:= dev_channel1
                    iosb:= iosbk,
                    p1 := buffer,p2:=512);
    {write the data to the remote node via VC }
    SendVC(m ,buffer, iosbk.count);
   end;
end.
```

```
program term_read(input, output);
    :
begin
    i:= Getsocket('TR    CSV3');     { get a local socket }
    j:= Findremotesocket('SILO  CSV3',i);  { get a remote socket }
    m:= OpenVC(i,j, 15);

    SendVC( m, buffer, 1):
    while forever do
     begin
       state:= $QIOW(func:=int(uor(io$_readvblk,io$m_noecho)),
                     chan:= device_channel2,
                     p1 := buffer,
                     p2:= 1 );
      {write the data to remote node via VC}
      SendVC(m, buffer,1);
    end;
end.
```

```
program SILO(input,Output);
    :
begin
    :
   i:= Getsocket('SILO    CSV3');
   j:=Listen(i,CVS3 );
   while forever do
    begin
      RcvVC(j,buffer,size);
     ( Do a Network Write to Remote Terminal Driver )
     status:= $qiow(func:=int(uor((uor(io$_writevblk,
                         io$m_network)),io$m_extend),
                   chan:= device_channel1,
   end;            p1 :=buffer, p2:=1 );
end.
```

CSV2 - Remote Node

CSV3 - Host node

Fig. 3.7 Actual Code of Processes in Fig. 3.6

at the remote node.

(2) to release the terminal line by request of the user.

(3) to regulate the total number of virtual terminal lines that a host node can allow.

The Terminal Server is implemented as two background processes on both the remote and host node. These processes, the TERM_CALLER at the remote node, and TERM_CALLEE at the host node, use the IPC Datagram service for Inter-process communication. There is no need to use the Virtual Circuit service since inter-process communication is required only intermittently - during line setup and release. To use virtual circuits, say in an n node network would require a node to maintain n-1 virtual circuits to the other n-1 nodes. This would certainly be a waste of resources.

To best explain the design of TERM_CALLEE and TERM_CALLER, the following sections will first discuss how a Virtual Terminal line is set up and how such a line is released. The last section formalizes the design in terms of the Virtual Terminal Protocol. The tables that are provided in that section encapsulates the rules and conventions used to communicate between the server processes and the subprocesses that they spawn per virtual terminal line.

3.4.1 Setting Up a Virtual Terminal Line

On the request of the user at the remote node, the Terminal Server processes - TERM_CALLER and TERM_CALLEE - cooperate in order to set up a Virtual Terminal line. Setting up such a line requires the Terminal Server processes to create the subprocesses: UART_COMBINE, UART and SILO at the host node and the TERM_READ and TERM_WRITE at the remote node. In turn, it is the subprocess pairs: (TERM_WRITE,UART_COMBINE) and (TERM_WRITE,SILO) that set up a Virtual Circuit between the host and remote node. The pair of Virtual Circuits, together, constitutes a Virtual Terminal line. The following are the various concerns of Terminal Server processes when setting up a Virtual Terminal line:

(1) Naming of Local and Remote Socket Names between Terminal Server subprocesses:
In order for these subprocesses to set up the two Virtual Circuits per Virtual Terminal line, they need to establish a unique local socket name and find a unique remote socket name per Virtual Circuit. Now

since the Remote Terminal Access design should allow multiple Virtual Terminal lines to emanate from a remote node, and conversely multiple Virtual Terminal lines to converge at any host node, the naming of local and remote sockets must reflect this capability. In other words, the name of the remote and local socket is dependent on both the name of host node and the specific Virtual Terminal line that it is granted. In this implementation, the local and remote socket names are a concatenation of such a combination.

(2) The order which the sub-processes are created:
The order in which the subprocesses are created is a function of the the IPC Virtual Circuit feature and the role of each individual subprocess. For example, looking at the partner processes TERM_READ (at the remote node) and SILO (at the host node), it is the TERM_READ process that receives a byte from the keyboard and sends the byte using SendVC service to the SILO process (recall Fig. 3.6). Therefore, the SILO process is the one that performs the Listen service, and since IPC requires a process that uses the Listen service to be created first, SILO must be created before TERM_READ. Similarly, it is the UART_COMBINE process (at the host node) that receives a string of

bytes from a local mailbox and writes to its partner TERM_WRITE via the IPC SendVC service. Therefore, the creation of TERM_WRITE must precede UART_COMBINE. Now the question still remains on which pair of subprocesses should be created first - (TERM_READ,SILO) or (TERM_WRITE,UART_COMBINE)? Since the remote user must be guaranteed an output for every input from his keyboard, the creation of TERM_WRITE should precede TERM_READ, lest the user initially hits the carriage return key when TERM_WRITE is not yet created. This would result in lost output which should have appeared at the remote screen. So the final conclusion is that TERM_WRITE-UART_COMBINE should be set up first. Fig. 3.8A and Fig. 3.8B describes the sequence of events that take place when a line is set up.

3.4.2 Releasing a Virtual Terminal Line

The escape sequence ESC] B is entered by the remote user when he wishes to release the virtual terminal line. To effect this, the Terminal Server processes must delete the two sub-process pairs. Unlike the setting of a virtual terminal line, the order of processes is unimportant. The only concern here is that the line is released so that TERM_CALLEE (at the host node) is able to reissue that line

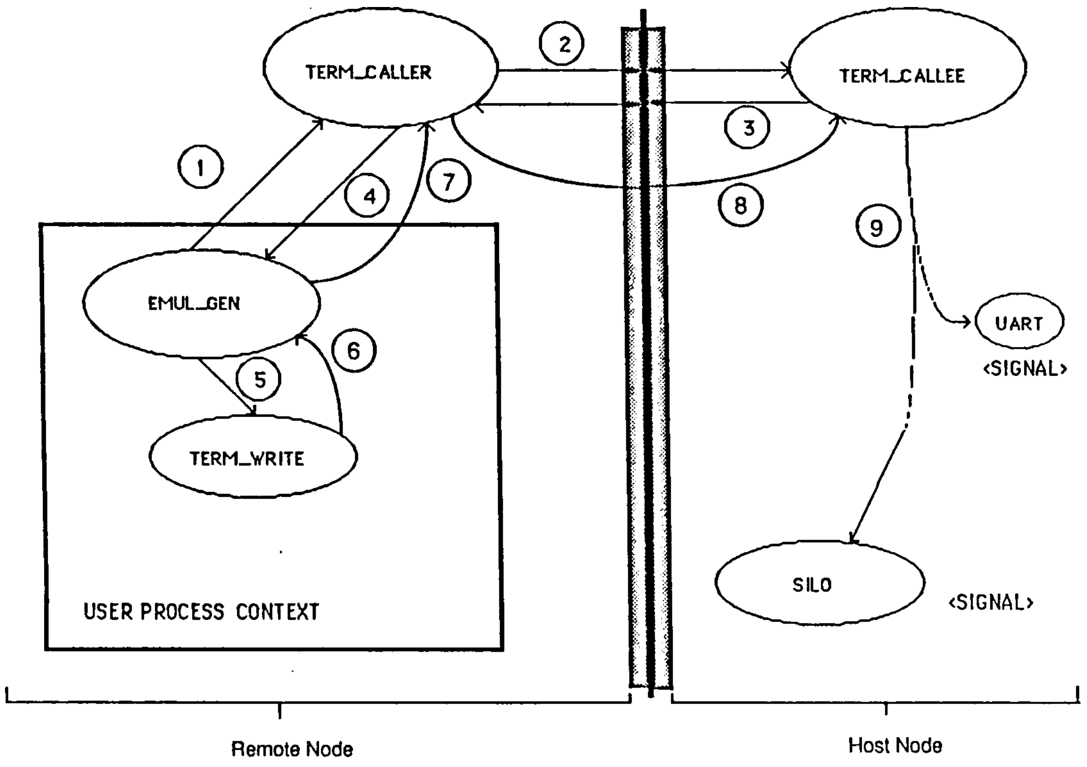

Fig. 3.8A Setting up a virtual terminal line

1 The User requests remote login to a specified remote node. EMUL_GEN (emulator generator) is activated on the user's behalf. EMUL_GEN then sends a message to the local terminal server TERM_CALLER to initiate the virtual line request.
2 TERM_CALLER sends a message to the remote terminal server TERM_CALLEE at the specified remote node.
3 TERM_CALLEE will search for a free virtual line. If there are none, it will send a "Bind Reject" message back to the originating TERM_CALLER. If there is an available line, TERM_CALLEE sends that line number to the originating TERM_CALLER.
4 TERM_CALLER relays the line number received to the originating EMUL_GEN (i.e. the originating user)
5 EMUL_GEN creates the subprocess TERM_WRITE on the user's behalf.
6 TERM_WRITE acknowledges its creation to EMUL_GEN.
7 EMUL_GEN, in turn, acknowledges TERM_WRITE's creation to TERM_CALLER.
8 TERM_CALLER, in turn, acknowledges TERM_WRITE's creation to TERM_CALLEE.
9 TERM_CALLEE proceeds to create the UART and SILO processes. It will wait until all the created subprocesses acknowledge their creation before continuing. See Fig. 3.8B.

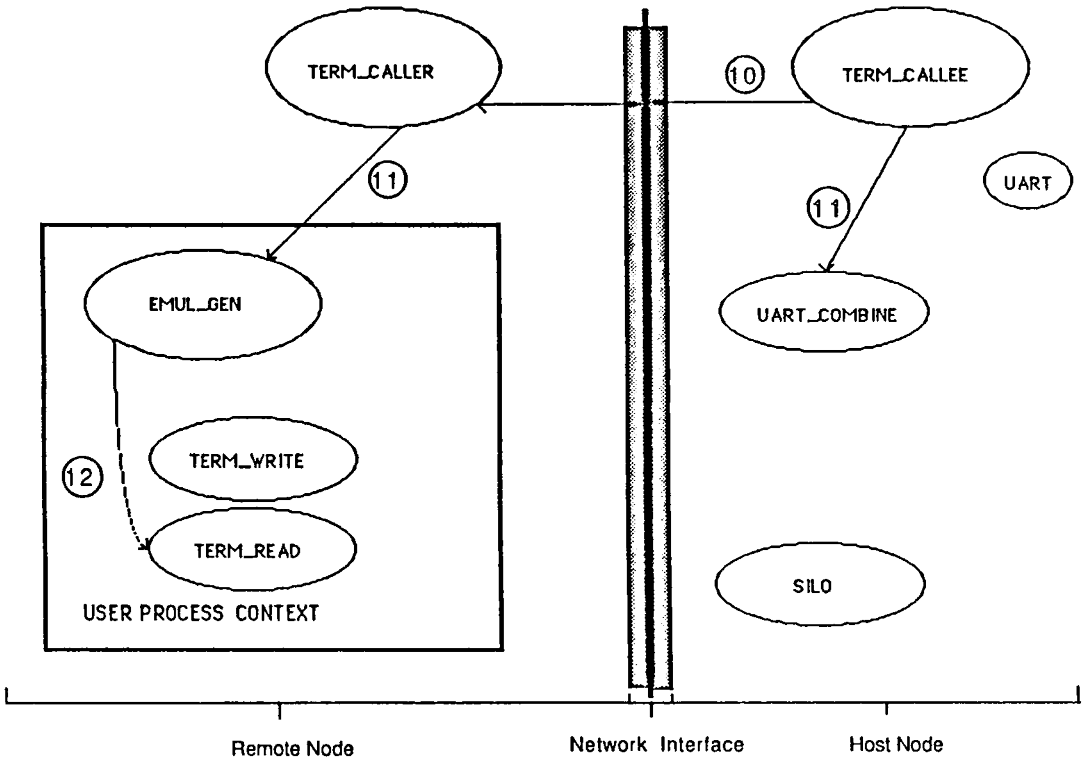

Fig. 3.8B Setting up a virtual terminal line

10 TERM_CALLEE, continues execution by sending a second message to the originating TERM_CALLER.

11 TERM_CALLEE proceeds to create the UART_COMBINE subprocess.

11 TERM_CALLER, on receiving the message from TERM_CALLEE, will notify the originating EMUL_GEN process.

12 EMUL_GEN creates the subprocess TERM_READ on the user's behalf. It will send a carriage return via IPC to initiate the remote login.

Note: The events represented by 11 and 11 can occur concurrently.

at some later request. Fig. 3.9 describes the sequence of events that take place when a line is released. It should be noted that only the sub-process pairs - (TERM_WRITE,UART_COMBINE) and (TERM_READ,SILO) are actually deleted.

3.5 Virtual Terminal Protocol

As we have seen, the Terminal Server Processes and its spawned subprocesses communicate across the network using certain rules and conventions; they constitute the Virtual Terminal Protocol. It is noteworthy to point out that the IPC primitives themselves do not form the Virtual Terminal Protocol. The protocol is formed by the specific rules and conventions in the naming and addressing of socket names, in the content and size of message fields, and the fixed order by which certain primitives are used.

In an effort to formalize the protocol, it is described in tables in terms of the types of messages, their functions and message formats. In addition, supplementary tables describe the sequence of message exchange that occur over time t. This approach is based on DECNET's formalization of its Network Virtual Terminal Protocol [VAXMAN G].

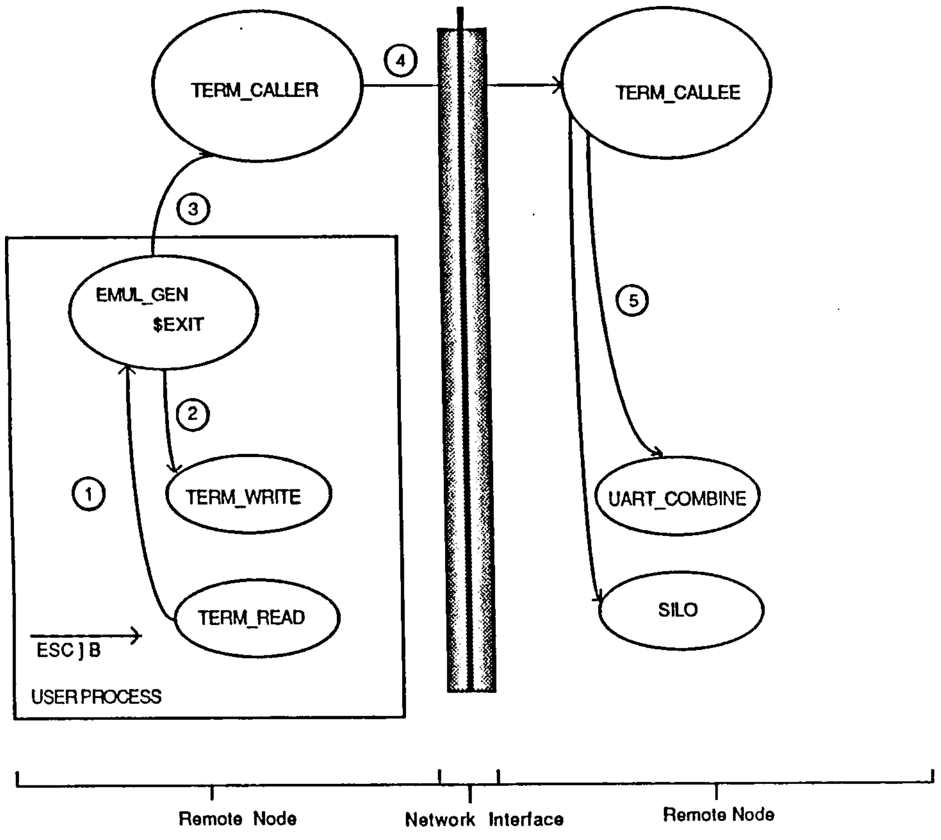

Fig. 3.9 Releasing a virtual terminal line

1. The user enters the escape sequence to initiate the line release. TERM_READ sends a message to EMUL_GEN and dies.
2. EMUL_GEN then notifies TERM_WRITE to "commit suicide".
3. EMUL_GEN then notifies TERM_CALLER to initiate line release at the remote node and then "commits suicide".
4. TERM_CALLER notifies TERM_CALLEE at the host node via IPC.
5. TERM_CALLEE notifies UART_COMBINE and SILO subprocesses to "commit suicide".

The Virtual Terminal Protocol can be subdivided into two categories: the protocol which is observed by the Terminal Server processes - called the Server Protocol - and the protocol which is observed by the spawned subprocesses - called the Virtual Line protocol.

Table 3 shows the Server Protocol message types, their functions, the originating process initiating the request and the message format. It should be noted that all message types are implemented as IPC communication primitives. Fig. 3.10A then describes a successful binding sequence. An unsuccessful binding sequence is describes in Fig, 3.10B. Similarly, the Virtual Line Protocol is described by Table 4 and Fig. 3.11.

3.6 Flow Control

End-to-end flow control in the RTA is enforced by:

(1) The XON/XOFF signals from the Remote Terminal's screen hardware to Driver:

The Terminal hardware stores incoming characters in a 64-character buffer and processes them in a FIFO basis. When the buffer content reaches 32 characters, the

Message	Function	Source	IPC Format
Bind Request	Requests a binding; identifies the specific remote node and specific process.	TERM_CALLER	buffer own_mbx source_node_name SendDG(i,j,buffer,16)
Bind Accept	Accepts a Bind Request; identifies the granted virtual line and the remote process that originated the call.	TERM_CALLEE	buffer own_mbx line_number SendDG(i,j,buffer,16)
Bind Ack	Acknowledges the Bind Accept	TERM_CALLER	"+" ack_message SendDG(i,j,buffer,8)
Bind Reject	Rejects a virtual line	TERM_CALLEE	buffer own_mbx "0" line_number SendDG(i,j,buffer,16)
Unbind	Releases a virtual line	TERM_CALLER	buffer line_number "-" kill_message SendDG(i,j,buffer,16)

Table 3 Terminal Server Protocol

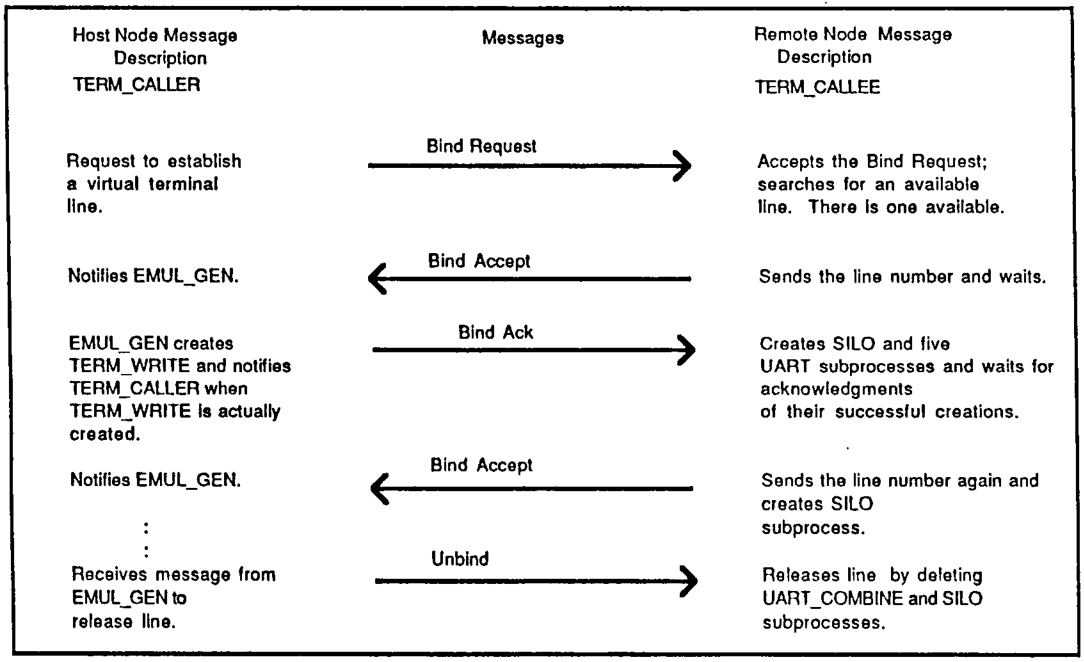
Host Node Message Description
TERM_CALLER
Messages
Remote Node Message Description
TERM_CALLEE
Request to establish a virtual terminal line.
Bind Request
Accepts the Bind Request; searches for an available line. There is one available.
Notifies EMUL_GEN.
Bind Accept
Sends the line number and waits.
EMUL_GEN creates TERM_WRITE and notifies TERM_CALLER when TERM_WRITE is actually created.
Bind Ack
Creates SILO and five UART subprocesses and waits for acknowledgments of their successful creations.
Notifies EMUL_GEN.
Bind Accept
Sends the line number again and creates SILO subprocess.
:
:
Receives message from EMUL_GEN to release line.
Unbind
Releases line by deleting UART_COMBINE and SILO subprocesses.

Fig. 3.10A A successful binding sequence

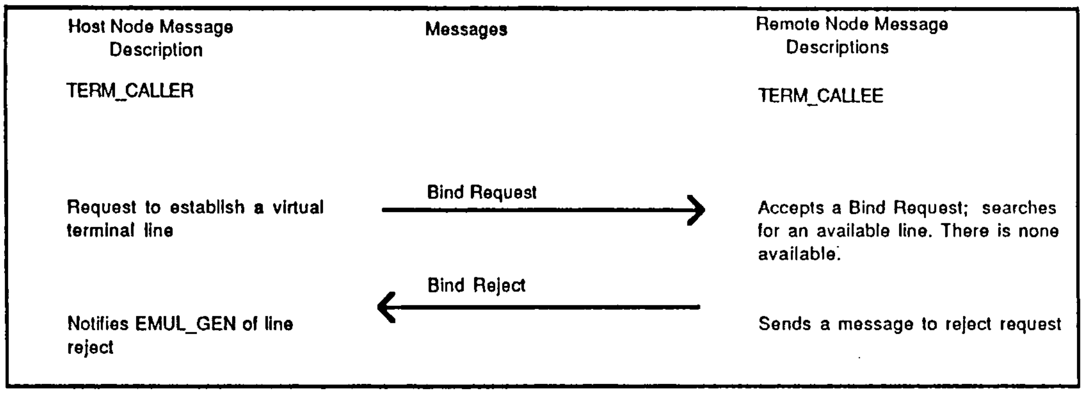
Host Node Message Description
TERM_CALLER
Messages
Remote Node Message Descriptions
TERM_CALLEE
Request to establish a virtual terminal line
Bind Request
Accepts a Bind Request; searches for an available line. There is none available.
Bind Reject
Notifies EMUL_GEN of line reject
Sends a message to reject request

Fig. 3.10B An unsuccessful binding sequence

Message	Function	Source	Destination	IPC Format
Send byte	Sends a byte of data	TERM_READ	SILO	□ buffer SendVC(m,buffer,1)
Receive byte	Receives a byte of data	SILO	TERM_READ	□ buffer RcvVC(m,buffer,1)
Send String	Sends a string of bytes of variable length	UART_COMBINE	TERM_WRITE	▭ buffer SendVC(m,buffer,size)
Receive String	Receives a string of bytes of variable length	TERM_WRITE	UART_COMBINE	▭ buffer RcvVC(m,buffer, size)

Table 4 Virtual Line Protocol

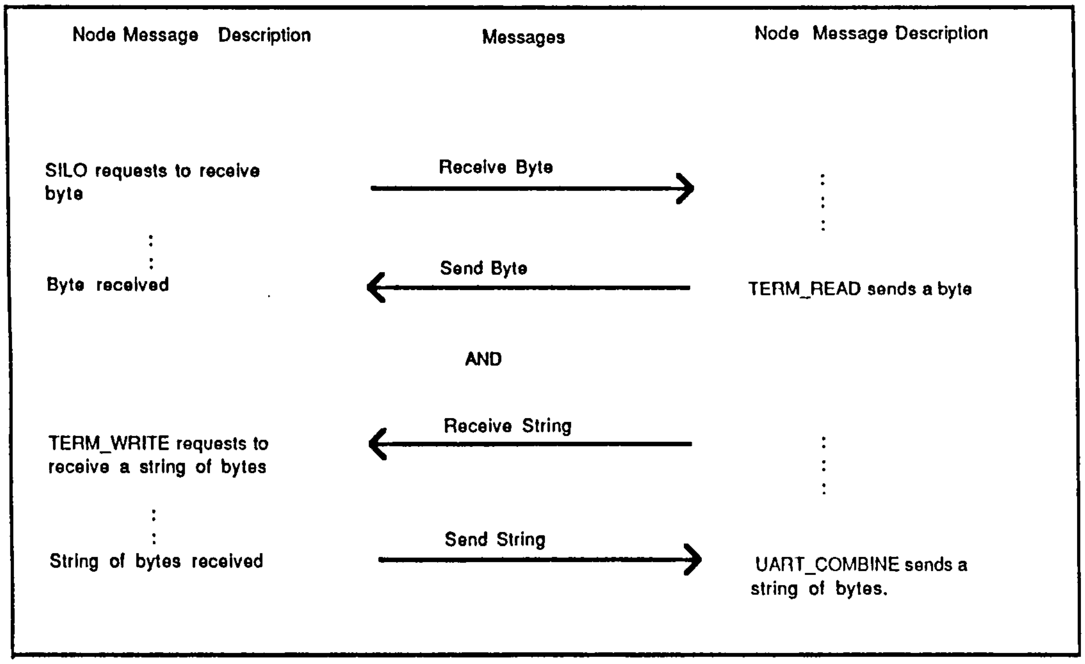

Fig. 3.11 Interprocess Communication on a Virtual Line

terminal will transmit XOFF (octal 023). On this signal, the host stops transmission to the terminal. Eventually, if the host stops transmitting, the terminal will deplete the buffer. When 16 characters remain in the buffer the terminal will transmit XON (octal 028) to signal the host that it may resume transmission. The Virtual Terminal unit's TTSYNC characteristic must be set for the driver to respond to the XON/XOFF signal.

(2) The XON/XOFF signals from the Driver to Remote Keyboard:
When the typeahead buffer for the Virtual Terminal is full, the driver sends a XOFF character to the Remote Keyboard. The keyboard hardware will then "freeze" all user input until the typeahead buffer clears. Then the driver will send a XON to the remote keyboard to free the keyboard. The Virtual Terminal unit's HOSTSYNC characteristic must be set for the keyboard to respond to the XON/XOFF signal.

(3) The inherent FIFO nature $QIO and the added stop-and-wait of $QIOW system service:
The VMS $QIO system service allows a user to initiate an I/O operation by queuing a request to the device's

associated driver. Once the I/O operation has been initiated, control will be returned to the user. The total size of the buffering, and the maximum size per message are adjustable $QIO parameters. The user then has the option of carrying on with the next instruction or it can wait for the I/O operation to complete. The $QIOW service caters to this feature. The FIFO nature of $QIO and the additional stop-and-wait feature of $QIOW are utilized by UART_COMBINE, the UART and SILO processes at the Remote Node - see Fig. 3.12.

The UART_COMBINE does a $QIOW Read from a mailbox and waits for input. When it finally reads a message from the mailbox, it will write this message to the remote TERM_WRITE via the SendVC primitive. Therefore, UART_COMBINE never has to go into a busy loop to wait for input from the mailbox.

Now the purpose of the UART process is to wait for input from the terminal driver. On the arrival of input, UART writes the string to mailbox and queues another $QIOW Network Read as soon as possible. Therefore, UART does a $QIO Write to the Mailbox and does not bother to wait for I/O completion to the mailbox.

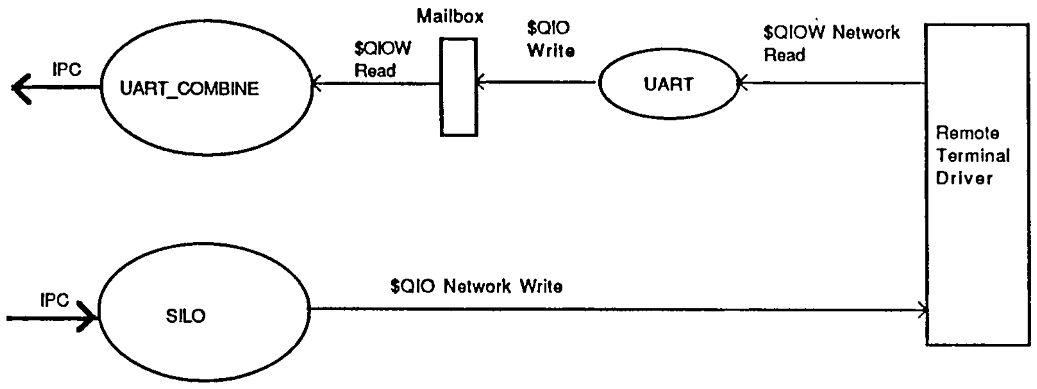

Fig. 3.12 Using the $QIO for flow control

The purpose of SILO, on receiving a byte via the RcvVC IPC primitive, is to write as quickly as possible to the terminal driver ("Input Interrupt") and return to wait for the next byte from the network. Therefore, it makes sense to do a $QIO to the Terminal Driver and go back to wait for the byte input from the network.

(4) The arrangement of IPLs within the Driver:
The arrangement of IPLs as described in Sections 3.2.1 and 3.2.2 allow the simulation of interrupts by using the Network $QIO Reads and Writes. However, as was pointed out, the flow control was not failsafe when system activity was high. Whereupon, the driver storage buffer (described in 3.2.1) was used to correct the situation.

CHAPTER 4

CONCLUSIONS

Of the three most popular application-level software services: Virtual Terminal, File Transfer and Electronic Mail, providing the Virtual Terminal feature is perhaps the most complicated of them all. While the implementation of File Transfer and Electronic Mail can be readily achieved by using programming techniques available to the general user, we have seen that providing a Virtual Terminal service requires programming a kernel-mode Virtual Terminal Driver at the host node.

Although the design of Remote Terminal Access relied on the IPC for network interprocess communication, the design and implementation of the Remote Terminal Driver itself is implementation independent. That is, the key feature of the design is the method by which the I/O from the external interface to the driver is no longer carried out by hardware interrupts but by regular system calls. Thus, the Virtual Terminal (or Remote Terminal Access) can be implemented on any network communication software. Another advantage of the design is that the implementation of a terminal server and terminal emulator modules at the remote node is relatively simple. The remote node need only to conform to the Virtual Terminal Protocol to set up the virtual circuits and remote terminal access would be a reality. Yet another advantage, and this is the cornerstone of this implementation, is the full DEC VAX/VMS terminal compatability of remote terminal driver.

However, the design is not without its shortcomings. The empirical solution of using a storage buffer to the problem of "overtaking" (Section 3.2.1) does not guarantee, under very heavy loads, that all output will appear at the screen of the remote terminal. While we could increase the storage buffer size, the tradeoff is the preservation of system dynamic memory versus the importance of guaranteed output. Given the infrequency of the "overtaking" condition culminating in storage buffer overflow, the preservation of system dynamic memory would seem to be the overriding concern. This is especially true since the increase in buffer space per unit is multiplied by the number of remote terminal units that are made available for remote login. Another shortcoming, perhaps a more serious one, is the inability of TERM_CALLEE to kill the UART process once it has started (Section 3.2.1), and as already mentioned, a guaranteed, cleaner method requires further knowledge of restricted VMS system routines.

Yet another shortcoming is that the speed of the virtual terminal line is comparable to that of a 1200 baud line. The main reason for this slowness in response is that the virtual line is a full-duplex line. That means that any character typed at the remote keyboard travels round-trip to the host system and back to the remote CRT. An improvement in responsiveness would entail the development of a "front-end" line processor at the remote node that would be constantly aware of the mode the remote user is in. For example, if the user is in the command line mode (VMS DCL), the "front-end" line processor would local-echo all characters and send the string to the terminal driver at the host node only when a carriage return key is entered. However, the line-processor would revert back to a full-duplex line when the user is in the VMS Editor and other modes that have byte oriented inputs. The complexity of "front-end" line processor is substantial, considering the out-of-band Control-Y, command recall, line editing and other features that are available in the VMS DCL interpreter. In addition, there would be some overhead involved in having the "front end" line processor having to constantly keep tab of the current user mode.

Since the Remote Terminal Access, in its current form, is usable and since all these shortcomings are rectifiable, the design of the Remote terminal Access was a very worthwhile project.

4.1 Summary

The topic of thesis concentrates on the design and implementation of remote terminal access from any computer to a VAX/VMS system. The only prerequisite condition is that the computer - the remote node - which accesses the VAX/VMS node - the host node - be on the same heterogenous network system based on the ETHERNET standard.

The strength of this design is a fully VMS compatible terminal driver that provides all the screen-oriented capabilities of a DEC VAX/VMS Terminal Driver. Also, its design involves terminal server functions to manage the total number of remote logons, and to set up and release virtual terminal lines.

4.2 Future Extensions

We suggest the following future extensions:

(1) Crash recovery

In the current implementation, crash-recovery, in both the IPC and the RTA, is not taken into consideration. In the event of a crash, especially during the setting of the virtual terminal line, the terminal server on the working node will get "hung-up". If the node was the remote node, this would prevent users at that node from remotely accessing to other host nodes that may be alive. Since crash recovery, in its strictest sense, should be the responsibility of the underlying network communication software, a future extension would be to implement a foolproof crash recovery scheme on the IPC. Then, the design of the RTA should be modified to take advantage of IPC's recovery scheme.

(2) Virtual Terminal Service to other systems.

A future extension could be to develop Virtual Terminal service to a UNIX system based on the RTA design on IPC. The existing RTA design could be used to set up and release virtual terminal lines. However, a kernel-based Virtual Terminal Driver will have to be implemented on UNIX system.

REFERENCES

[GIGI] GIGI Terminal Installation and Owner's Manual, page 60.

[HIL86] Wael Hilal, "Interconnecting Two Ethernets", to be published, 1987

[IPC] Chao C.P. , "Design and Implementation of Interprocess Communication Procedures on Ethernets."

[UNIX] Maurice J. Bach, The Design of the UNIX Operating System, page 329, Prentice-Hall, 1986.

[VAXMAN A] VAX/VMS I/O User's Reference Manual, Part 1, page 8-2.

[VAXMAN B] VAX/VMS I/O User's Reference Manual, Part 1, Appendix A.

[VAXMAN C] VAX/VMS Writing a Device Driver, pages 5-1 - 5-17.

[VAXMAN D] VAX/VMS Writing a Device Driver, pages 1-2 - 1-3.

[VAXMAN E] VAX/VMS Writing a Device Driver, pages 1-5 - 1-6, pages A-1 - A-38.

[VAXMAN F] VAX/VMS Writing a Device Driver, pages 8-3 - 8-13.

[VAXMAN G] Digital network architecture (phase IV) General Description, page 6-7 - 6-12.

[VAXMAN H] VAX/VMS Writing a Device Driver, pages 14-1 - 14-18.

[VAX 84A] Lawrence J. Kenah, Simon F. Bate, VAX/VMS Internals and Data Structures, page 7, Digital Press, 1984.

[VAX 84B] Lawrence J. Kenah, Simon F. Bate, VAX/VMS Internals and Data Structures, pages 30 - 41, Digital Press, 1984.

[VAX 84C] Lawrence J. Kenah, Simon F. Bate, VAX/VMS

Internals and Data Structures, page 431, Digital Press, 1984.

[VT100] VT100 User Guide, page 87.