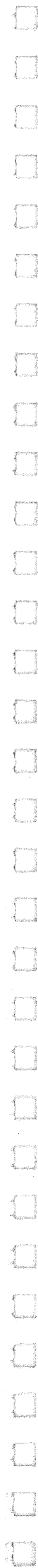
EDUCATIONAL SERVICES

VAX/VMS Training

VAX/VMS Device Driver Related Topics

digital



EY-2278E-MJ-0001

RELATED TOPICS

Prepared by Educational Services
of
Digital Equipment Corporation

First Edition, October 1984

Copyright © 1984 by Digital Equipment Corporation
All Rights Reserved

The reproduction of this material, in part or whole, is strictly prohibited. For copy information, contact the Educational Services Department, Digital Equipment Corporation, Bedford, Massachusetts 01730.

Printed in U.S.A.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may not be used or copied except in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

The manuscript for this book was created using DIGITAL Standard Runoff. Book production was done by Educational Services Development and Publishing in Nashua, NH.

The following are trademarks of Digital Equipment Corporation:

digital™	DECtape	Rainbow
DATATRIEVE	DECUS	RSTS
DEC	DECwriter	RSX
DECmate	DIBOL	UNIBUS
DECnet	MASSBUS	VAX
DECset	PDP	VMS
DECsystem-10	P/OS	VT
DECSYSTEM-20	Professional	Work Processor

CONTENTS

INTRODUCTION 10-3
OBJECTIVES 10-3
RESOURCES 10-4
LEARNING ACTIVITIES 10-4
TOPICS 10-5

- Ancillary Control Processes (ACPs) 10-6
- Existing ACPs 10-7
- ACP Interfacing Components 10-7
- MOUNT Image 10-8
- ACP Queue Block (AQB) 10-10
- Volume Control Block (VCB) 10-11
- Window Control Block (WCB) - Disk Only 10-11
- File Control Block (FCB) - Disk Only 10-11
- Consistency Checking by MOUNT Image and ACP 10-12
- ACP Initialization and Operation 10-13
- $QIO System Service Operation 10-15
- Dismount Image 10-15
- Pseudo-Devices 10-16
- Example Use of Pseudo-Device 10-17

XQPS 10-19

- XQP Overview 10-19
- XQP Caching 10-19
- XQP Lock Manager Usage 10-20
- XQP Startup 10-21
- XQP Merged into Pl Space 10-21
- XQP One-time Initialization 10-22
- $QIO System Service Operation for XQPs 10-24
- Dispatching the Request 10-25

CLASS/PORT DRIVERS 10-26

- Terminal I/O 10-28

INSTRUCTION SET 10-29
WRITING A DISK DRIVER 10-32
ATTENTION ASTS 10-34

- Driver Setup 10-35
- Defining Listheads 10-36
- Routine COM$SETATTNAST 10-36
- Driver Interrupt Service Routine 10-38
- Routine COM$DELATTNAST 10-38
- Canceling Attention ASTs 10-39
- Coding FORTRAN AST Routines 10-40

CONNECT-TO-INTERRUPT 10-41

- Preparing for Connect-to-Interrupt 10-42
- System Manager Responsibilities 10-42

Connecting to an Interrupt 10-43
Mapping UNIBUS I/O Space 10-44
Connect-to-Interrupt $QIO Parameters 10-45
Specifying User-Supplied Driver Subroutines 10-47
Specifying Event Flag Setting 10-48
Shared Buffer 10-49
User-Supplied Driver Subroutines 10-50
Connecting a Process to a Device 10-51
Starting the Device 10-52
Recovering from Power Failure 10-53
Servicing Interrupts 10-54
Canceling I/O Requests 10-58

FIGURES

10-1 ACP-$QIO Interaction 10-6
10-2 Database Formed for an ACP 10-9
10-3 Fields that Form an AQB 10-10
10-4 Control Flow for Pseudo-Devices 10-18
10-5 Pl Space 10-23
10-6 Class/Port Drivers 10-26
10-7 Class Driver Request Packet (CDRP) 10-27
10-8 Class Driver, Port Driver and Associated UCB 10-28
10-9 General Format of Device Table in Module DEVTAB of MOUNT, and Size of Fields 10-33
10-10 Format of ACB/Fork Block Created by COM$SETATTNAST 10-37
10-11 Double-Mapping Shared Buffer 10-49

EXAMPLES

10-1 Flow of Instructions in the Main Loop of All ACBs 10-14

INTRODUCTION

A number of areas within the operating system require the presence of a device driver, or influence the actions of a device driver. This module examines some of those areas, including ACPs and XQPs.

The terminal driver on VAX/VMS is divided into two pieces, the terminal port driver and the terminal class driver. These are discussed briefly. This module also discusses some of the techniques used in the executive to produce efficient kernel mode code, and considerations to make regarding which of several instruction sequences to use.

In addition, drivers may be written that allow privileged user programs to control hardware activity on devices (often useful in real-time applications). Two mechanisms are discussed that provide this ability: set attention ASTs, and the connect-to-interrupt system service.

OBJECTIVES

Upon completion of this module, you will be able to:

1. State the steps needed to write an ACP that interacts with the $QIO system service.
2. Describe the function of Files-11, ODS-2 XQP.
3. Describe the features of the terminal's class and port drivers.
4. List the components of the executive that must be modified when writing a device driver for disks.
5. Recognize efficient coding techniques within the executive.
6. Write a driver that uses the set-attention-AST feature.
7. Write interrupt-servicing routines that can be declared using the connect-to-interrupt system service.

RESOURCES

1. VAX/VMS File-Structured Devices Reference Manual
2. VAX/VMS System Services Reference Manual

LEARNING ACTIVITIES

1. Study existing drivers incorporating the set-attention-AST feature. Study the source listings in module COMDRVSUB of the executive for the routines that implement the set-attention-AST scheme.
2. Study existing drivers and ACPs that interact to perform I/O operations.
3. Study existing drivers and XQPs that interact to perform I/O operations.
4. Study existing disk drivers, and modules INIT and MOUNT in the executive source listings.

TOPICS

- ACPs
  - Uses
  - Functions
  - Data structures
- XQPs
- Class/Port drivers
- Terminal drivers
- Instruction set notes
- Writing a disk driver
- Attention ASTs
- Connecting to interrupts

Ancillary Control Processes (ACPs)

ACPs

- Interface between user's $QIO request and driver.
- Perform supplemental functions.
- FDT routines decide whether to queue the I/O request to the ACP or to the driver.
- Are created when a program (MOUNT image) is run on the system, usually at system startup time (CMKRNL, MOUNT, DETACH privileges required).
- Are processes because they have:
  - the ability to invoke system services.
  - their own privileges, quotas, limits.
  - their own file protection.
  - their own scheduling priority.
- Are needed if:
  - Preprocessing of the I/O request requires calling system services (not allowed in FDT routines since services can raise or lower IPL).
  - Nonprivileged processes want the ability to perform privileged operations (for example, privilege to perform physical I/O).

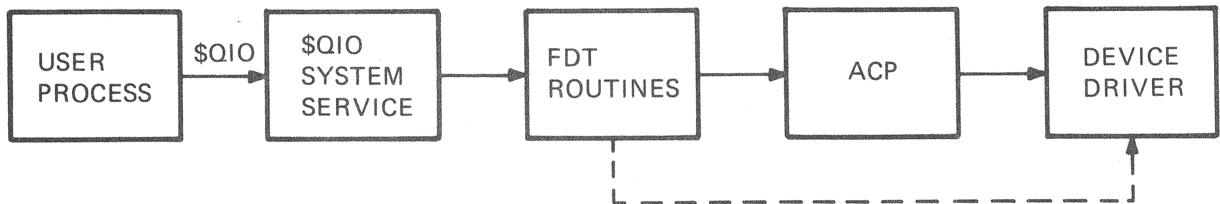


Figure 10-1 ACP-$QIO Interaction

Existing ACPs

- Disk (Files-11, ODS-1 only)
- Magnetic Tape
- Network
- Remote Terminal

ACP Interfacing Components

- MOUNT Image
  - Establishes I/O database
  - Creates ACP process
- ACP process itself
- $QIO system service and FDT routines
  - Preprocess I/O request
  - Queue I/O request to ACP
- DISMOUNT Image
  - Breaks ACP communication paths
  - Deallocates ACP internal database

MOUNT Image

The basic functions of the MOUNT image are:

- Creates data structures for ACP
- Creates ACP process by means of $CREPRC
- Issues $QIO with IO$_MOUNT function code.

When the MOUNT image is run, it:

- Uses the device name and unit number
- Determines the file that contains ACP image
- Assigns a process name to the ACP process
- Invokes $CMKRNL to change the access mode to kernel
- Allocates and initializes ACP data structures

MOUNT/PROCESSOR=option

Associate an ACP to process a volume, overriding default manner in which ACPs are assigned (OPER privilege required).

Option	Meaning
UNIQUE	Create a new process to execute copy of default ACP for specific device type or controller.
SAME:device	Use the ACP currently being used by the specified device.
file-spec	Create a new process to execute the ACP specified by the file spec (CMKRNL and OPER privileges required).

Initializing the ACP Database

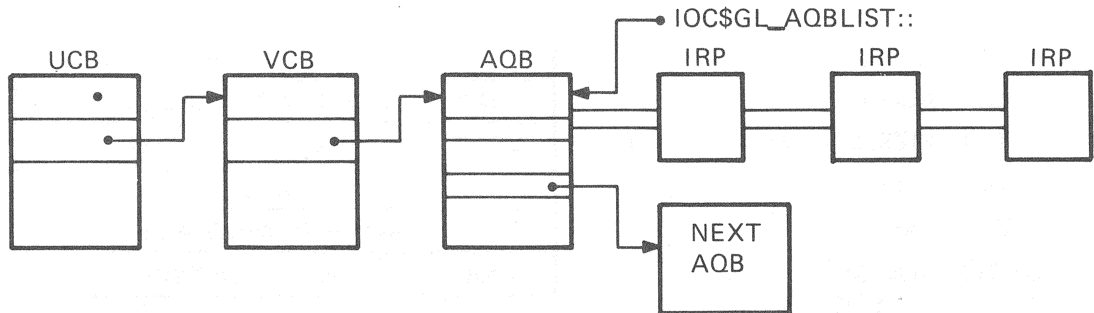


Figure 10-2 Database Formed for an ACP

- Major control blocks used by ACP are:
  - AQB (ACP Queue Block)
  - VCB (Volume Control Block)
  - UCB (Unit Control Block)
- Blocks are in nonpaged pool.
- Any AQB in a singly linked list of AQBs can be found using global symbol IOC$GL_AQBLIST.

MOUNT CREATING AQB

set Emier

MOUNTING UCB

$CREPRC ACP

QIOW IO$_MOUNT

ACP set MOUNTED bit in the UCB.

ACP Queue Block (AQB)

Field			
AQB$L_ACPQFL	IRP LISTHEAD FORWARD LINK		
AQB$L_ACPQBL	IRP LISTHEAD BACKWARD LINK		
AQB$W_SIZE, B_TYPE, B_MNTCNT	VOLS	TYPE	SIZE
AQB$L_ACPPID	ACP PROCESS ID		
AQB$L LINK	POINTER TO NEXT AQB		
AQB$B_STATUS, B_ACPTYPE, B_CLASS	SPARE	CLASS	ACP TYPE STATUS

MKV84-1889

Figure 10-3 Fields that Form an AQB

- Contains the listhead of IRPs awaiting service by the ACP.
- Size, type, and pointer fields are filled in.
- Count field (AQB$B_MNTCNT) is initialized to 1 to indicate that ACP is handling one volume.
- If a MOUNT image requests an existing ACP process, pointers are followed from UCB, to VCB, to AQB, then from AQB to AQB until match on process ID is found. Count field is then incremented.

Volume Control Block (VCB)

- Created/allocated from nonpaged pool after AQB is created or found.
- Describes a volume being serviced by the ACP.
- Address of VCB is stored in the UCB (UCB$L_VCB).
- Holds the address of the AQB in VCB$L_AQB.
- Holds parameters from volume's home block (disk) or label block (tape).
- Holds transaction count or number of I/O requests in progress for ACP.

Window Control Block (WCB) - Disk Only

- Contains a set of mapping pointers.
- Facilitates translation of virtual block numbers in a file to logical block numbers on a volume.

File Control Block (FCB) - Disk Only

- Controls simultaneous access to a file.

Consistency Checking by MOUNT Image and ACP

- Two bits are set by the MOUNT image and cleared by the ACP

  1. Creating bit in AQB (AQB$V_CREATING in AQB$B_STATUS)

     Checks whether created process is really an ACP

     Prevents two users from creating ACP simultaneously

  2. Mounting bit in UCB (UCB$V_MOUNTING in UCB$W_STS)

     Checks that ACP was not started with a RUN command

- MOUNT image does the following:

  - Sets consistency check bits
  - Issues $CREPRC for ACP, specifying file name, process name, quotas, and privileges
  - Places PID of created ACP (returned by $CREPRC) into AQB
  - Starts a timer (say 30 seconds)
  - Issues a $QIO with IO$_MOUNT function code
  - Waits while ACP or driver clears the bits, and ACP sets mounted bit in UCB (DEV$V_MNT in UCB$L_DEVCHAR)
  - After timer expires, examines creating bit in AQB

- If creating bit is:

  set

  - MOUNT image deallocates data structures, deletes ACP

  clear

  - MOUNT image exits

- Mounting bit

  - If clear when ACP called, ACP deletes itself
  - Usually cleared by driver as part of mount $QIO processing

- MOUNT image must be careful to return to nonpaged pool the AQB and VCB data structures if it deletes the ACP process.
- MOUNT image synchronizes with the system while accessing I/O database
  - Obtains/releases I/O database mutex (IOC$GL_MUTEX)
  - Uses system routines SCH$LOCKW and SCH$UNLOCK
  - These routines leave process at elevated IPL (SYNCH or ASTDEL)

ACP Initialization and Operation

- ACP locates and stores AQB address
  - Performs $GETJPI to find its own PID
  - Follows AQB pointers, starting at IOC$GL_AQBLIST, looking for PID match
  - AQB address is needed to access IRPs queued off AQB
- ACP clears creating bit in AQB
- Mounting bit is cleared when $QIO with IO$_MOUNT serviced
- Mounted bit in UCB device characteristics longword is set
- ACP enters main loop, hibernates until IRP to process
- Awakened by EXE$QIOACPPKT when IRP placed in AQB queue
- ACP processes request, then hibernates again
- After processing an IRP, ACP checks whether volume should be dismounted
  - Dismounts volume if transaction count is one, and volume is marked for dismount.
  - As each volume is dismounted, count field (AQB$B_MNTCNT) is decremented.
  - When count is zero, ACP deallocates VCB and AQB data structures and deletes itself.

```
ACP_MAIN:
        $HIBER_S                        ; wait for IRP
    10$:
        REMQUE @aqbaddr,packet          ; get next IRP
        BVS ACP_MAIN                    ; if vs, none in queue
              .
              .
              .
        process IRP
              .
              .
              .
        update transaction count and
        if transaction count does not equal 1,
        then go to 10$
              .
              .
              .
        BBC dismount, unit, 10$         ; if clear, then
              .                         ; no dismount pending
              .
              .
        attempt to dismount volume
              .
              .
              .
```

Example 10-1   Flow of Instructions in the
Main Loop of All ACBs

$QIO System Service Operation

- $QIO system service invokes FDT routines to fill device-dependent portions of IRP.

- FDT routines determine need to call the ACP.

- If ACP needed, FDT routines

  - raise IPL to IPL$_SYNCH

  - queue IRP to ACP instead of to driver

    JMP G^EXE$QIOACPPKT

    This queues IRP to appropriate ACP by following pointers from UCB to VCB to AQP.

- Available FDT routines (ACP-related) - to use these, driver and ACP must understand complex-chained buffers (CXBs).

ACP$READBLK	(all read functions for disk and tape)
ACP$WRITEBLK	(all write functions for disk and tape)
ACP$MOUNT	(service IO$_MOUNT $QIO)
ACP$ACCESS	(used for IO$_ACCESS and IO$_CREATE)
ACP$DEACCESS	(used for IO$_DEACCESS)
ACP$MODIFY	(used for IO$_ACPCONTROL, IO$_DELETE, IO$_MODIFY)

- FDT routines should increase volume transaction count (VCB$W_TRANS) for each IRP queued to ACP. ACP decreases count after I/O operation is completed.

Dismount Image

- Marks a volume for dismount

  - DEV$V_DMT bit in UCB$L_DEVCHAR is set

- May issue $QIO request to ACP for volume dismount

- Runs in kernel mode

- Dismount operation

  - performed by ACP
  - occurs when volume transaction count has gone to one
  - occurs when volume is marked for dismount

Pseudo-Devices

- Used when
  - Both user and ACP must $ASSIGN a channel to a non-shareable device because both issue $QIOs to the device.
  - More than one user needs to share a non-shareable device (for example, DECnet links sharing a DMR).
- User assigns pseudo-device, issues $QIO to it.
- Driver for pseudo-device contains only FDT routines, and passes IRPs to the ACP.
- ACP assigns real device and issues $QIO to it.

Example Use of Pseudo-Device

User assigns pseudo-device YY for which driver is YYDRIVER. ACP assigns real device XX for which driver is XXDRIVER.

1. XXDRIVER and YYDRIVER are loaded.
2. ACP (say YYACP) is mounted.
3. YYACP assigns a channel to device XX.
4. A user assigns a channel to device YY.
5. The user issues a $QIO to device YY.
6. YYDRIVER validates request, passes IRP to the ACP.
7. The ACP examines the IRP:
   - Determines the function code.
   - Copies necessary data into its buffers.
   - Issues a $QIO to device XX. ACP may request physical I/O whereas user requested virtual I/O.
8. XXDRIVER initiates device XX to service request.
9. The ACP's $QIO completes. The ACP
   - May copy data into nonpaged pool to return to user.
   - May update pointers in user IRP to point to data in nonpaged pool.
   - Copies I/O status block from its $QIO request into user's IRP: IRP$L_MEDIA and IRP$L_MEDIA4 (returned as the IOSB from the user $QIO).
   - Updates UCB operation count and VCB transaction count for YY.
   - Queues the user IRP to IOPOST.
10. The user $QIO completes.

Steps 4 through 10 can be repeated for the same (or a different) user.

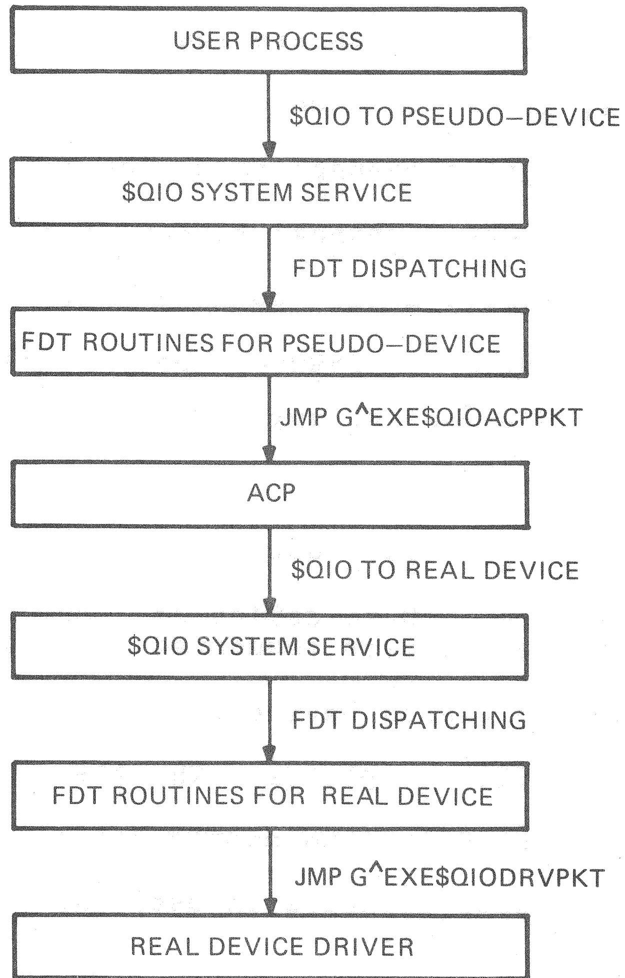


Figure 10-4 Control Flow for Pseudo-Devices

XQPS

XQP Overview

- Used instead of ACPs for Files-11, ODS-2 Disk Structures.
- Functionally equivalent to an ACP.
- Not a separate process, but part of process that issues $QIO.
- Mapped into P1 space of a process when process is created.
- Installed shareable.
- Caching is done on a system-wide basis.
- Lock manager is used to coordinate file access.
- Process start-up time is increased due to mapping of additional global sections and initializing XQP in P1 space.

XQP Caching

- XQP has full caching.
- Buffer space for the XQP is initialized in P1 space.

XQP Lock Manager Usage

- XQP uses two kinds of locks:
  1. synchronization lock
  2. access lock
- Synchronization lock is requested prior to performing any file manipulations to maintain the file's integrity.
- Access lock allows manipulation of the file.
- No locks are charged to the process.
- RMS file opens typically require a directory lookup. The major I/O function (IO$_ACCESS) is accompanied by a directory lookup subfunction.
- File opens with directory lookup require:
  - A synchronization lock/unlock and an access lock/unlock on the target directory for the directory lookup.
  - A synchronization lock/unlock and an access lock for the file itself.
- File closes require a synchronization lock/unlock and an access unlock.
- Locks used to protect/control cache usage.

XQP Startup

XQP Merged into P1 Space

- Merged by XQPMERGE, a routine called by PROCSTRT.
- Number of global sections to map specified by XQP$W_SECTIONS.
- Section names are SYSQXP_nnn (nnn=Ø to (XQP$W_SECTIONS - 1)).
- Permanent portion of P1 is expanded to accommodate the merged image.
- CTL$GL_F11BXQP in P1 pointer page is used to point to XQP, and initially points to base address of XQP.
- XQPMERGE jumps to INITXQP code (SYSXQP_ØØØ) where the XQP one-time initialization code is performed.

XQP One-time Initialization

- There are three buffer pools in paged system pool, each consisting of a virtually contiguous area:

  1. Storage Bitmap (allocated first)
  2. Directory, Random Data Block, Quota File
  3. File Header, Index File

- A private kernel stack is created and locked into the working set.
- A pure code segment is locked into the working set.
- An impure data segment is locked into the working set.
- The XQP queue heads are initialized.
- The XQP dispatch address, DISPATCH in module DISPATCH, is set up.
- CTL$GL_F11BXQP in the P1 pointer page is set to point to the address of the XQP's work queue.

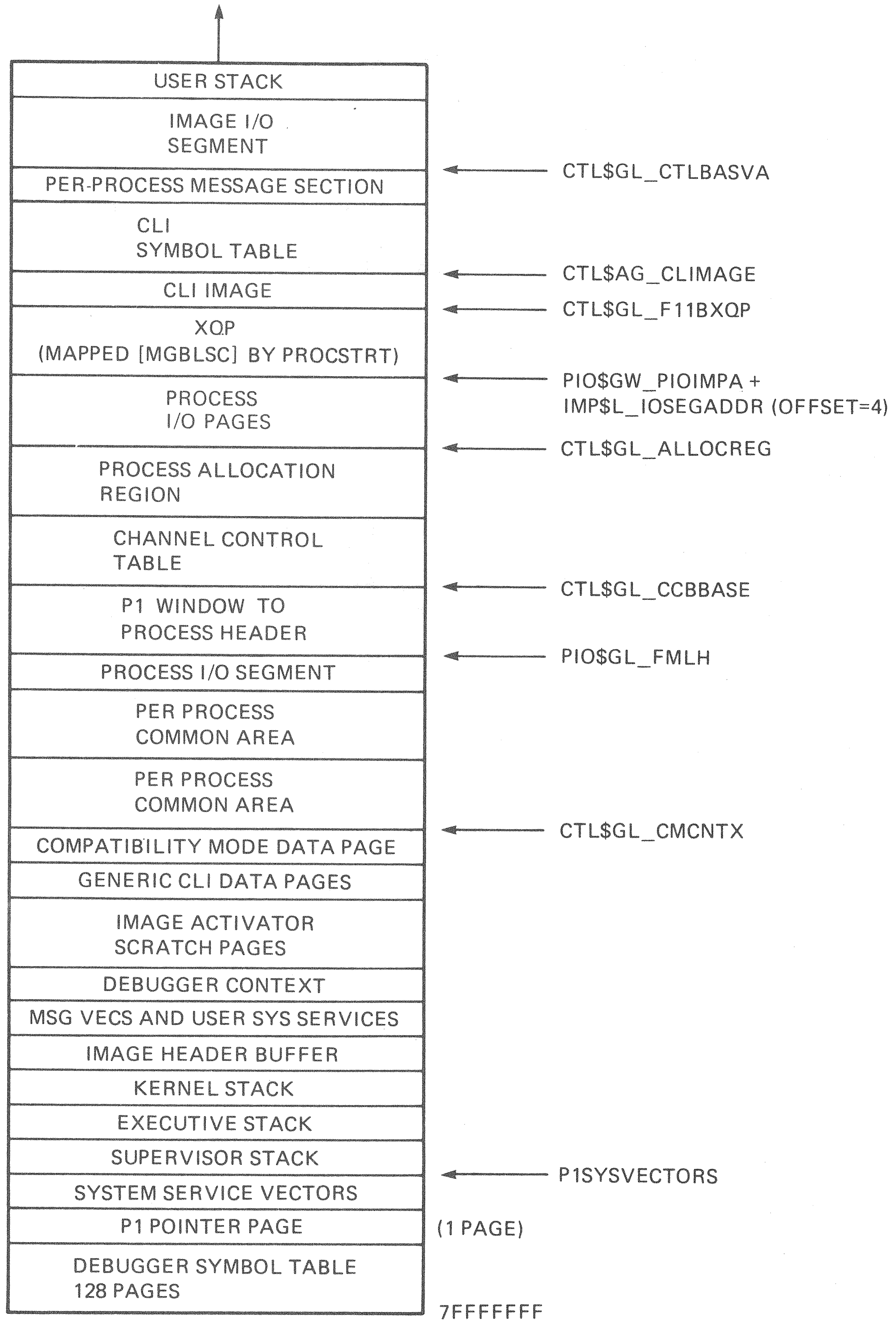


Figure 10-5 P1 Space

$QIO System Service Operation for XQPs

- $QIO system service invokes FDT routines to fill device-dependent portions of IRP.
- FDT routines determine the need to send IRP to ACP/XQP.
- If ACP and XQP is needed, FDT routines raise IPL to IPL$_SYNCH, then

  JMP G_^EXE$QIOACPPKT (old routine)

  or

  JMP G_^EXE$QXQPPKT (new routine)
- In routine EXE$QIOACPPKT, pointers are followed from the UCB to the VCB to the AQB. This is the normal ACP case. Then the ACP PID in AQB$L_ACPPID is tested against zero (0). If 0, this is an XQP, so branch to EXE$QXQPPKT.
- In EXE$QXQPPKT, set up an AST control block and queue a kernel mode AST to DISPATCH routine of XQP.

Dispatching the Request

- IRP is queued to XQP work queue.
  - If an operation is in progress, RET to user.
  - If no operation in progress, then:

    Disallow process deletion

    Switch to private kernel stack

    Call main routine DISPATCHER

- In main routine DISPATCHER:
  - Enable main condition handler.
  - Dequeue an IRP.
  - Initiate measurements (disk reads, CPU time, and so on).
  - If physical read/write, return.
  - Perform $QIO function. Major functions are:

    IO$_ACCESS IO$_CREATE IO$_DEACCESS
    IO$_DELETE IO$_MODIFY IO$_ACPCONTROL IO$_MOUNT

  - If function completed successfully, call cleanup routine. This checks whether buffers were written, and performs necessary flushes.
  - Calculate final measurement counters (disk reads, CPU time, and so on).
  - Call IO_DONE in module IODONE to post I/O completion for the request.
  - Release XQP synchronization lock.

CLASS/PORT DRIVERS

- Class Drivers
  - Handle I/O for each class of devices (disks, tapes, and so on).
- Port Drivers
  - Handle I/O for the specific PORT (communications) devices.
  - Class/Port Interface

    Allows class and port drivers to communicate without extra coding effort. Makes the task of adding hardware easier.

    When a different communications/controller device is added, the port driver is the only additional software needed.

    A Class Driver Request Packet (CDRP) is used to pass additional information between the class and port drivers.

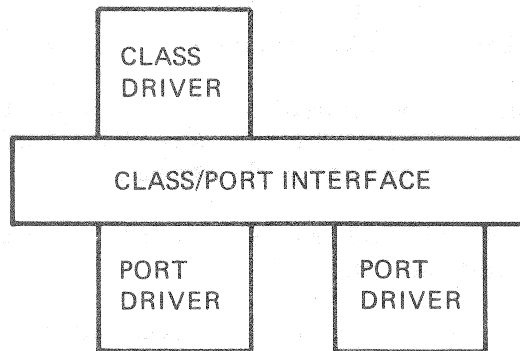


Figure 10-6 Class/Port Drivers

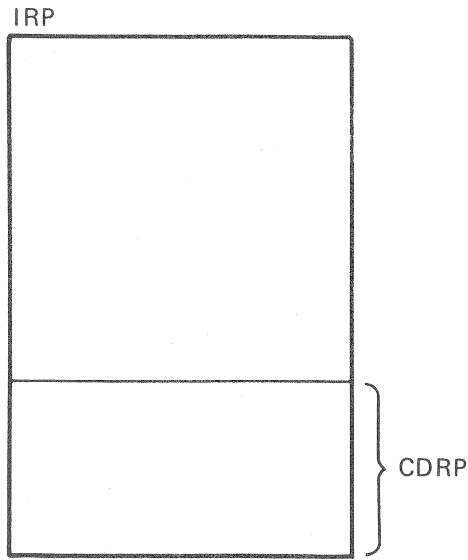


Figure 10-7 Class Driver Request Packet (CDRP)

Terminal I/O

The terminal I/O interface uses class/port style coding but without the formal CLASS/PORT interface.

Class Driver	TTDRIVER	
Port Drivers	DZDRIVER	(DZ-11,DZ-32)
	CONINTDSP	(within exec, for console)
	YCDRIVER	(DMF-32)

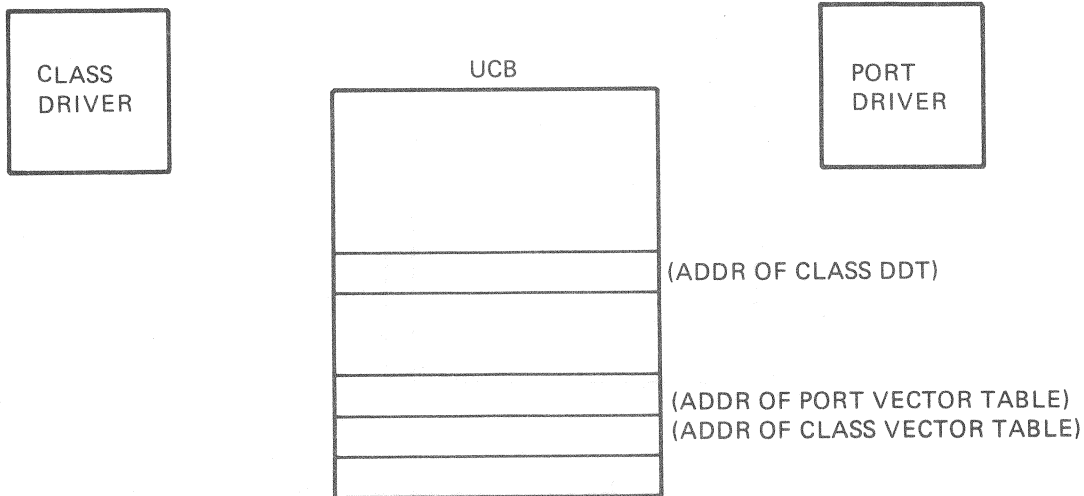


Figure 10-8 Class Driver, Port Driver and Associated UCB

1. Class and Port Vector Table - similar to DDT. Contains address of routines used for specific purposes.

2. Port Unit Initialization Routine:

   - Gets address of Class Driver DPT (stored in TTY$GL_DPT)

   - Fills in UCB with Vector Table address for Class and Port drivers (vector address stored in DPT)

   - Places address of Class DDT into UCB

INSTRUCTION SET

1. Align data in natural order (not across longword boundaries): Quad|Long|Word|Byte

   - Up to 2-3 times faster execution speed.
   - Fetch/write longword in one memory access.
   - FORTRAN naturally aligns data.
   - In FORTRAN COMMON blocks, avoid misaligning data.

   ```
   COMMON /x/ ABYTE,LONG,IWORD,JWORD
   ```

   is better stored as

   ```
   COMMON /x/ LONG,IWORD,JWORD,ABYTE
   ```

2. Overlap memory writes with other instructions.

   - register-register instructions
   - I/O page references (in driver)

   (See UBA interrupt service routine and change mode dispatcher EXE$CMODKRNL.)

3. Bit field instruction tradeoffs:

   BITW, BISW, BICW are generally faster than BBS, BBC

   however, BIxx usually take more bytes than BBx.

Number of bytes required by BITW for bit number operand:

Testing Bit #	# Bytes for Operand
Ø-5	1
6-7	2 (immediate mode used)
8-15	3
16-31	5 (immediate mode used)

Number of bytes required by BBS for bit number operand = 1 since bit number can always be expressed as a short literal. An 8-byte instruction buffer prefetches instructions. Using 5 bytes for just one operand reduces the decoding effectiveness of this buffer.

4. Saving registers:

   - Use MOVQ Reg, -(SP) for two consecutive registers.

   - Using PUSHR #^M<>, the microcode does a longword move for each register after scanning the mask.

   - For RØ-R5, the mask fits into a short literal, so PUSHR may be more efficient than several MOVX instructions.

5. Adding

   - Can use MOVAx with displacement addressing off a register:

     MOVAB X(R1), R1

     R1 = R1 + X

   - Indexed addressing can also be used:

     X + [1 / 2 / 4 / 8] * register

     MOVAL @#6[R1], R1

     R1 = R1*4 + 6

6. To set/clear a single bit when there is no mask:

   - BBS bit-number, destination, 10$
   - 10$: next-instruction
   - BIS, BIC instructions need a mask.

7. Using JMP @(R5)+ versus JMP @(R5)

   - JMP @(R5)+ is slightly faster and occupies 2 bytes of memory. Use when you are not concerned that register (R5) is incremented.
   - JMP @(R5) occupies 3 bytes of memory; @(R5) is assembled as @B^0(R5).

8. To set two or more bits in a device register:

   - Set the bits in a system register.
   - Use one MOVx instruction rather than two or more BISx instructions.
   - Data and acknowledgement signals must travel over the SBI and UNIBUS/MASSBUS once only.

WRITING A DISK DRIVER

1. Borrow as much code as possible from existing UNIBUS/MASSBUS drives.

2. Implement the ODS-1 or ODS-2 file system.

3. Define the disk geometry in the disk extension to the UCB (number of sectors, tracks, cylinders).

4. ACP --> Nothing needs to be changed.

5. INIT and MOUNT may require some modification.

   There is an identical table in both INIT and MOUNT (Figure 10-9 in module DEVTAB), associated with MOUNT in the microfiche. To see if INIT and MOUNT require modification:

   - Determine whether table specifies disk for your driver (use sector, track, cylinder fields).

   - If DECtype disk exists, call your disk by corresponding disk type in the comment.

   - If disk does not exist, patch the table (both INIT and MOUNT) to contain entries for your device (see VAX-11 Patch Utility Reference Manual).

   - Blocking factor = 512/(#bytes/block). This is usually 1.

     If factor is not 1, some VMS utilities will not work.

SECTOR	TRACK	CYLINDER	SIZE	BLOCKING FACTOR	DEFAULT CLUSTER SIZE	DEFAULT MAX. # OF FILES	FLAGS	COMMENTS (DISK TYPE)
26	1	77	494	4	1	123	DVT_NOGEOM	; FLOPPY
22	3	411	27126	1	2	4000	DVT_FACTBAD	; RK06
32	5	823	131680	1	6	15000	DVT_FACTBAD	; RM02/03
22	19	815	340670	1	11	25000	--	; RP06
0	0	0	0	0	0	0	0	; SPARE
BYTE	BYTE	WORD	LONG-WORD	BYTE	WORD	LONGWORD	BYTE	NOT APPLICABLE

MKV84-1887

Figure 10-9 General Format of Device Table in Module DEVTAB of MOUNT, and Size of Fields

ATTENTION ASTS

- Mechanism used in VMS to notify a process that some device condition has occurred.
- Allow high-level language (e.g., FORTRAN) AST routines to control a device.
- Requested through $QIO (normally IO$_SETMODE function).
- AST is delivered when device condition occurs.
- Requires no outstanding $QIO request.
- Saves the overhead involved in processing a special kernel AST, and I/O post processing.
- Must be reenabled after AST delivered.

Examples

- Terminal Driver
  - Control-C AST
  - Control-Y AST
  - Control-anything AST
- Mailbox driver
  - Write Attention
  - Read Attention
- DMC11 Driver
  - Waiting Message
  - Error Condition

Driver Setup

- UCB must contain a listhead for each device condition.
- Each device condition matches a legal attention AST.
- Listheads are defined using $DEFINI, $DEF, and $DEFEND.
- Determine which function codes to use:

Normally used	Function modifiers
IO$_SETMODE	IO$M_ATTNAST
IO$_SENSEMODE	IO$M_READATTN
IO$_SETCHAR	IO$M_WRTATTN
IO$_SENSECHAR	

- Place function codes in FDT table using the FUNCTAB macro.
- In FDT routines, you must:
  - Check accessibility of P1 and P2 parameters.

    P1 = address of AST routine, or
       = Ø (to disable previous requests).
    P2 = AST parameter (optional).
    P3 = access mode (optional) at which AST delivered.

  - Place address of appropriate listhead in R7 using MOVAL, not MOVL.

    ```
    MOVAL UCB$L_LISTHEAD(R5), R7
    ```

  - Call system routine COM$SETATTNAST.

    ```
    JSB G^COM$SETATTNAST
    ```

  - Exit the FDT with either:

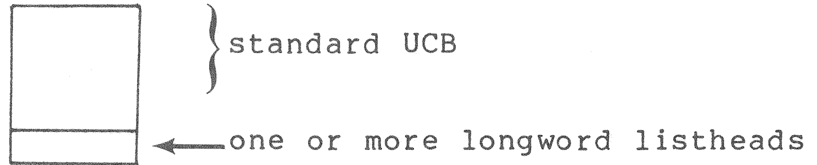
    JMP G_^EXE$FINISHIO — writes R0/R1 in IOSB

    or

    JMP G_^EXE$FINISHIOC writes R0 clears R1 in IOSB

    Do not use JMP G^EXE$QIODRVPKT because the $QIO call used to request the AST does not remain outstanding; it must finish immediately.

Defining Listheads




```
$DEFINI UCB
 .=UCB$K_LENGTH                     ; position to end of UCB
$DEF UCB$L_HEAD1   .BLKL 1          ; first listhead
$DEF UCB$L_HEAD2   .BLKL 1          ; second listhead, etc.
$DEF UCB$K_MYLEN                    ; length of extended UCB
$DEFEND UCB
```

Routine COM$SETATTNAST

- Checks process AST quota.
- Allocates and builds AST Control Block (ACB) that also functions as a fork block.
- Raises IPL to UCB$B_DIPL (working previously at IPL = 2) by means of DSBINT macro.
- Inserts ACB in a singly linked list identified by the listhead in UCB (whose address was placed in R7).
- Lowers IPL by means of ENBINT macro to the previous level.
- Returns.

The data structure in Figure 10-10 serves as both a fork block and an AST control block. The FIPL used is 6.

DOUBLY_LINKED LIST		
FIPL	TYPE	SIZE
FORK PC		
FORK R3		
FORK R4		
P1 = AST ROUTINE ADDRESS		
P2 = AST PARAMETER		
CHANNEL	UNUSED	RMOD
PID OF REQUESTING PROCESS		

MKV84-1891

Figure 10-10 Format of ACB/Fork Block Created by COM$SETATTNAST

Driver Interrupt Service Routine

- Must restore owner UCB address in R5.
- Must place address of listhead in R4 using MOVAL (not MOVL).

  MOVAL UCB$L_LISTHEAD(R5), R4

- Call routine COM$DELATTNAST be means of JSB.

  JSB G^COM$DELATTNAST

- Dismiss the interrupt by restoring registers RØ-R5, and issue an REI instruction.
- Note that bit UCB$V_INT in UCB$W_STS (an expected interrupt) is not set by COM$SETATTNAST, since there is no outstanding $QIO as seen by the routine.

Routine COM$DELATTNAST

- For each ACB/fork block in list:
  - Remove it from list.
  - Fork on it (fill in PC,R3,R4).
- Fork routine (runs at IPL=6)
  - Rearranges the block into an ACB.
  - Jumps to SCH$QAST to queue the AST.
  - SCH$QAST raises IPL to IPL$_SYNCH for synchronization with rest of scheduling routines.

Canceling Attention ASTs

- User specifies AST routine address as zero (P1=Ø)
- FDT routine places address of listhead in R7

  ```
  MOVAL UCB$L_LISTHD(R5),R7
  ```

- FDT routine places channel number in R6 and calls COM$FLUSHATTNS

  ```
  MOVZWL IRP$W_CHAN(R3),R6
  JSB G^COM$FLUSHATTNS
  ```

  or

  FDT routine calls COM$SETATTNAST which calls COM$FLUSHATTNS if P1=Ø.

- COM$FLUSHATTNS
  - Raises to UCB$B_DIPL by means of DSBINT macro
  - For each ACB with matching PID and channel:

    Removes it from list

    Issues ENBINT macro

    Restores AST quota to process

    JSBs to COM$DRVDEALMEM to deallocate ACB

    Raises IPL to UCB$B_DIPL again by means of DBSINT
  - Returns

Coding FORTRAN AST Routines

- Use $QIOs rather than FORTRAN READ/WRITE statements.
- Reference device registers as word (INTEGER*2) or BYTE variables, never as INTEGER*4 variables.
- Return the CSR address to the user process (possibly in IOSB in an IO$_SENSECHAR $QIO). Then user can pass CSR address to AST routines that control the device by way of the AST parameter.
- Use %VAL to pass the CSR address to subroutines or AST routines. CSR is an INTEGER*4 variable that contains the address of the CSR.
- Use EQUIVALENCE statements to associate character strings and bytes. CHARACTER strings and INTEGER data must be in separate COMMON blocks.
- Reenable the attention AST after your AST is entered. A separate routine is needed to issue the $QIO, since FORTRAN does not allow a routine to specify itself as an AST routine.

CONNECT-TO-INTERRUPT

- Purpose
  - Allows a process to control a device.
  - Allows fastest interrupt response on VAX.
  - Allows preemption of other system processing to handle a real-time event in an arbitrarily complex manner.
  - Allows data to be buffered from a device in real-time, and later processed in process context.
- Possibilities
  - Have event flag set upon device interrupt
  - Execute AST procedure at every device interrupt
  - Execute user-supplied "driver subroutines" to:

    Start I/O on device

    Handle interrupts from device

    Cancel I/O on device

    Respond to powerfail recovery
- Restrictions
  - UNIBUS device only
  - Non-DMA device only (buffered I/O functions)
  - I/O cannot be timed (no timeout routine)
  - Non-shareable devices only
  - Single device unit per controller

Preparing for Connect-to-Interrupt

System Manager Responsibilities

- Reserve system page table entries for use by processes connecting to interrupts.

  ```
  $ RUN SYS$SYSTEM:SYSGEN

  SYSGEN> USE ACTIVE
  SYSGEN> SET REALTIME_SPTS desired_value
  SYSGEN> WRITE NEWFILE.PAR
  SYSGEN> WRITE CURRENT
  SYSGEN> EXIT
  ```

  Reboot system.

- Connect device in system using connect-to-interrupt driver.

  ```
  $ RUN SYS$SYSTEM:SYSGEN

  SYSGEN> CONNECT MYAØ/CSR=device_CSR/VECTOR= -
                  device_vector/ADAPTER=UBA_TR_# -
                  /DRIVER=CONINTERR
  ```

  Repeat command for each device (or device vector) for which you will allow a connect-to-interrupt. One device with multiple interrupt vectors will have multiple UCBs, one for each vector.

Connecting to an Interrupt

- Assign a channel to the device.
- Map UNIBUS I/O space containing device registers into process address space to control device from your process (in AST routine). Requires PFNMAP privilege.
- Issue a connect-to-interrupt $QIO request specifying:
  - An AST routine (maybe).
  - An event flag (maybe).
  - Which driver routines you are providing (if any).
  - Address of a data buffer shared by your process and your driver routines. Requires CMKRNL privilege.
- Control and/or respond to device interrupts, process data, etc.
- Disconnect from the interrupt by issuing a $CANCEL system service call over the channel on which the connect-to-interrupt is made.

Mapping UNIBUS I/O Space

- Use the $CRMPSC system service.
- Requires PFNMAP privilege.
- Most important arguments are:

INADR - identifies which region (P0 or P1) to map into.

RETADR - contains address range to which UNIBUS I/O space is mapped.

FLAGS - SEC$M_PFNMAP => mapping by page frame number.
SEC$M_WRT => device registers can be written.
SEC$M_EXPREG => expand region identified by INADR to accommodate mapped page(s).

PAGCNT - indicates how many pages are to be mapped. Specify more than one if your device registers cross a page boundary.

VBN - page frame number of first page to be mapped. To obtain this value, use the following:

```
$IO780DEF

PFN=<IO780$AL_UB0SP+^Odevice_CSR>@-9
```

IO780$AL_UB0SP is starting address of UNIBUS 0. Add to that the CSR for your device. Divide the address by 512 (@-9) to get a PFN.

Connect-to-Interrupt $QIO Parameters

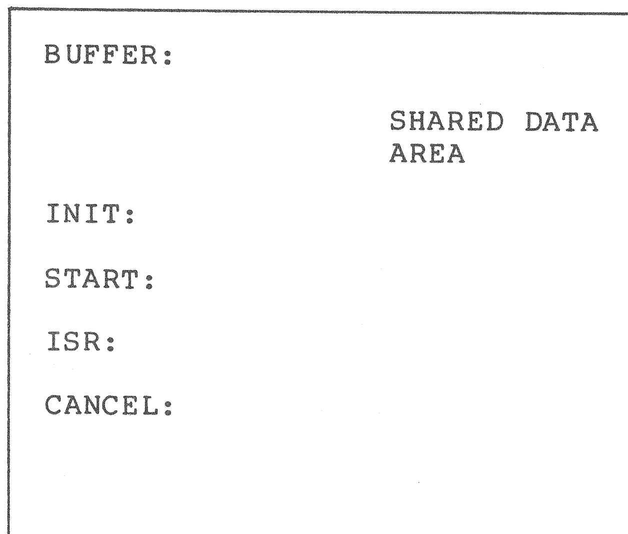
- Function codes (both use buffered I/O quota, BYTLYM)
  - IO$_CONINTREAD (shared buffer is read-only)
  - IO$_CONINTWRITE (buffer can be read/written)
- P1 is buffer descriptor for shared code/data buffer. First longword contains number of bytes in buffer, second longword contains buffer address. If buffer is shared, CMKRNL privilege is required.
- P2 is address of entry point list of four longwords containing offsets into shared buffer:
  - (0) CIN$L_INIDEV device initialization routine
  - (4) CIN$L_START start I/O routine
  - (8) CIN$L_ISR interrupt service routine
  - (12) CIN$L_CANCEL cancel I/O routine
- P3 is flags for options to the connect-to-interrupt facility:
  - CIN$M_EFN Set event flag on interrupt.
  - CIN$M_USECAL Use CALL interface to user routines rather than default JSB interface.
  - CIN$M_INIDEV Device initialization routine in buffer.
  - CIN$M_START Start I/O routine in buffer.
  - CIN$M_ISR Interrupt service routine in buffer.
  - CIN$M_CANCEL Cancel I/O routine in buffer.
  - CIN$V_EFNUM Starting bit number of event flag to set on interrupt.
  - CIN$S_EFNUM Start of event flag field.
  - CIN$M_REPEAT Repeat delivery of interrupts.

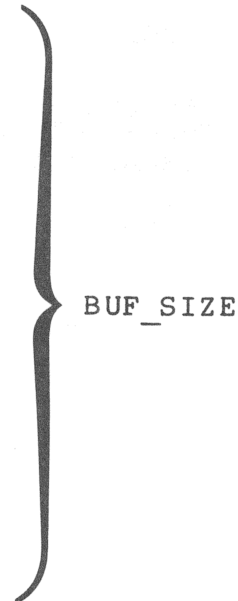
- P4 is address of entry mask of an AST routine called as result of interrupt.
- P5 is AST parameter. Used only if process-supplied interrupt service routine does not overwrite the value.
- P6 is number of AST control blocks to preallocate in anticipation of fast, recurrent interrupts from the device.

Specifying User-Supplied Driver Subroutines

```
$QIO_S      FUNC=#IO$_CONINTREAD, p1=BUF_DESC, -
            P2=#ENTRY_LIST, -
            P3=#<CIN$M_INIDEV!CIN$M_START! -
                 CIN$M_ISR!CIN$M_CANCEL>, - - -

BUF_DESC:
        .LONG BUF_SIZE
        .ADDRESS BUFFER

ENTRY_LIST:                         ; note offsets are used

.LONG INIT - BUFFER
.LONG START - BUFFER
.LONG ISR - BUFFER
.LONG CANCEL - BUFFER
```

SHARED BUFFER





Specifying Event Flag Setting

```
$QIO_S P3=#<CIN$M_EFN!<20@CIN$V_EFNUM>>, -

          . . .
```

Sets event flag 20 on each device interrupt

Specifying AST procedure

```
$QIO_S P4=#AST_PROC, P5=#6, P6=#3, -

          . . .

AST_PROC: .WORD  entry mask

          RET
```

AST_PROC is called after a device interrupt with an AST parameter of 6. Three AST control blocks are preallocated in anticipation of fast, recurrent interrupts.

Shared Buffer

- Locked in memory by CONINTERR.
- Double-mapped (see Figure 10-11)
  - In user process space
  - Through allocated SPTEs
- User-supplied driver subroutine addresses stored in UCB.
- Can be written if function code in connect-to-interrupt $QIO is IO$_CONINTWRITE.

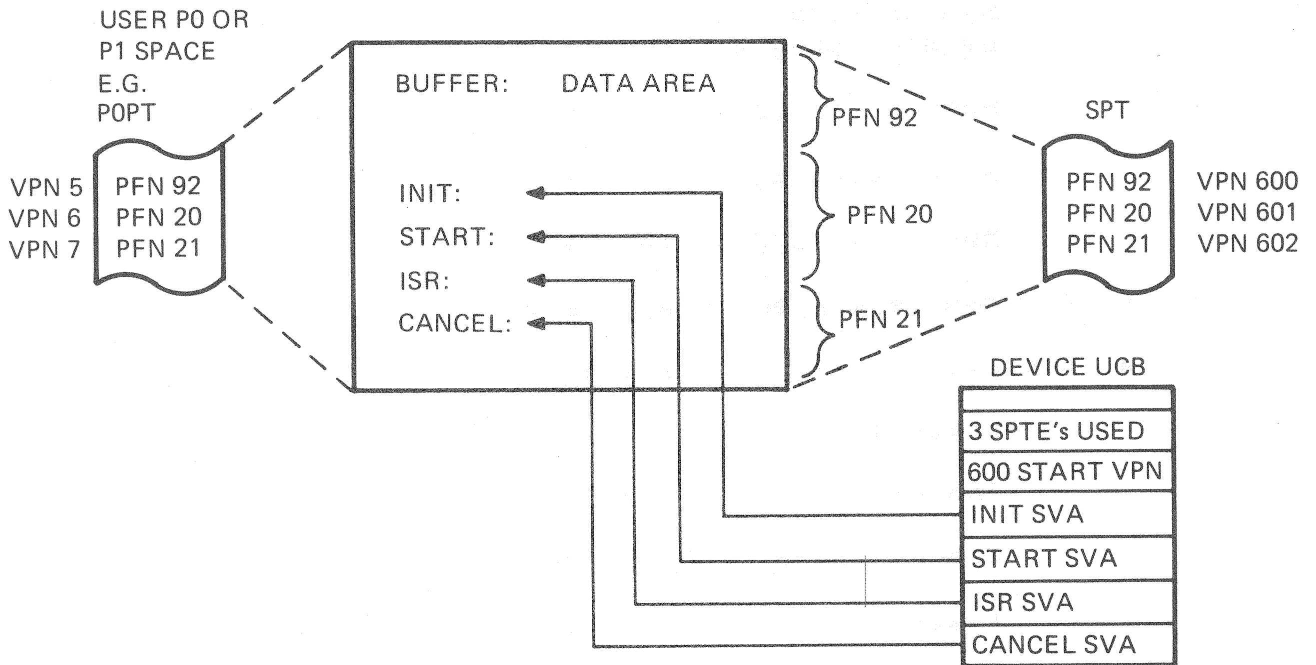


Figure 10-11 Double-Mapping Shared Buffer

User-Supplied Driver Subroutines

- Constraints
  - Must be position-independent code.
  - Must follow rules for accessing I/O space.
  - May only access:

    shared buffer area (double-mapped)

    system address space
  - Must not incur exceptions.
  - Synchronization of data in shared buffer must be done by user program.
  - Must not perform lengthy processing.
  - Must have registers used unless otherwise noted.
  - Must exit appropriately (RSB or RET).
- Are double-mapped in system space.
- Are executed in kernel mode.
  - Start I/O at IPL$_QUEUEAST (6)
  - ISR at device IPL (22)
  - Cancel I/O at IPL$_QUEUEAST (6)
  - Powerfail recovery at IPL$_POWER (31)

Connecting a Process to a Device

The FDT routine in the connect-to-interrupt driver, CONINTERR, connects process to device as follows:

1. Validates $QIO parameters and clears any event flag.

2. Locks the physical pages of shared buffer into memory and double maps the pages into system page table entries.

   - Pages accessed by process-supplied control routines if buffer length in descriptor is nonzero and process has CMKRNL privilege.

   - System manager must have reserved system page table entries using SYSGEN.

3. Constructs argument lists and calling interfaces to process-supplied routines.

   - Store values in the device's UCB.

   - Calculate routine addresses from system-mapped page table entries, not process buffer addresses.

4. Allocates specified number of AST control blocks to process.

   - Checks that process has enough AST quota for number of ASTs requested.

5. Inserts each AST control block in a queue in the device's UCB.

Starting the Device

1. CONINTERR FDT routine exits by means of EXE$QIODRVPKT.

2. EXE$QIODRVPKT then JSBs to the CONINTERR start I/O routine.

3. CONINTERR start I/O routine transfers control to the process-supplied start I/O routine using JSB (default) or CALL.

JSB Case:

- Before entering start I/O routine, address of argument block to be placed in R2.

- Address of process-supplied routine is in UCB$L_CI_START.

- When entered, process-supplied start I/O routine can expect:

  ```
  R2       = address of counted argument list
  R3       = address of IRP
  R5       = address of UCB

  Ø(AP)    = 4 (argument count)
  4(AP)    = system address of process buffer
  8(AP)    = address of IRP
  12(AP)   = system address of device's CSR
  16(AP)   = address of UCB
  ```

- Argument list, built by FDT routines, is located through UCB$L_STARGC.

- The process-supplied start I/O routine:

  - Is entered at IPL 6 (IPL$_QUEUEAST).

  - May raise but not lower IPL below 6 during processing.

  - Must preserve all registers except RØ-R2 and R4.

  - Must initiate and start activity on the device.

  - May enable interrupts by initializing a device-specific control register.

  - Accesses device registers as memory locations in UNIBUS I/O page.

- Locates addresses through offsets from CSR address in the IDB (found since UCB address is known).
- Must exit at IPL$_QUEUEAST using an RSB.

CALL Case:

- CONINTERR start I/O routine issues a CALLG to the process-supplied start I/O routine (found in UCB$L_CI_STACAL).
- Same inputs and restrictions apply as in JSB case.
- Routine should exit with a RET instruction.

Recovering from Power Failure

CONINTERR device initialization routine is called:

- Sets the device on-line (UCB$M_ONLINE in UCB$W_STS).
- Marks device as owner of controller (IDB$L_OWNER).
- Calls process-supplied device initialization routine with:

  JSB case:

  R0 = UCB address
  R4 = CSR address
  R5 = IDB address
  R6 = DDB address
  R8 = CRB address

  CALL case:

  0(AP) = 5 (argument count)
  4(AP) = CSR address
  8(AP) = IDB address
  12(AP) = DDB address
  16(AP) = CRB address
  20(AP) = UCB address

- Process-supplied routine should not change IPL (at 31 = IPL$POWER).
- All registers except R0-R3 must be preserved.

Servicing Interrupts

When device interrupts, CONINTERR interrupt service routine gains control and:

- Places IDB address in R4, and removes IDB pointer from stack.
- Places owner UCB address (IDB$L_OWNER) in R5.
- Places argument list address (UCB$L_CI_ISARGC) in R2.
- JSBs to process-supplied interrupt service routine, or in CALL case, JSBs to two-instruction routine in CONINTERR which executes CALLG to process-specified interrupt service routine, then RSBs.

The process-specified interrupt service routine:

- May expect the following when it executes

```
Ø(AP)    = 5 (argument count)
4(AP)    = system address of process buffer
8(AP)    = AST parameter address
12(AP)   = system address of CSR
16(AP)   = IDB address
2Ø(AP)   = UCB address

R2       = address of counted argument list
R4       = IDB address
R5       = UCB address
```

- May copy contents of device registers into the shared buffer double mapped into system page table entries (or into AST parameter).

- Cannot reference process virtual addresses since it executes in system context.

- Is responsible for clearing error and interrupt bits in device registers, and/or restarting the device.

- Must maintain an IPL equal to, or greater than, device IPL (22).

- Must save and restore all registers other than RØ-R5 used in routine.

- Must restore stack to its original state before exiting.

- Must load interrupt status into RØ:

  1 => queue AST and set event flag if either or both were specified.

  Ø => do not notify the process of the interrupt.

- Must exit with a RET (CALL case) or RSB (JSB case).

After return to CONINTERR interrupt service routine:

- If RØ is not equal to 1, RØ-R5 are restored, followed by REI.
- If RØ is equal to 1, routine:
  - Dequeues an AST control block from UCB queue (if user requested it).
  - Creates a fork process.
- AST control block is used as fork block if available, otherwise UCB is used as fork block.
- Flag is set to indicate no further forks possible (UCB$V_CI_UCBFRK in UCB$L_DEVDEPEND).
- Future attempted forks are not performed; interrupt is dismissed with RØ-R5 restored before the REI is issued.

Fork process gains control at IPL 6 (IPL$_QUEUEAST) and:

- Places the address of the PCB in R4.
- If the fork block is the UCB:
  - the error code SS$_DISCONNECT is placed in RØ
  - the process-supplied cancel I/O routine is called
  - the process is disconnected from the interrupt
- If an AST is requested, the AST is queued to the process by means of SCH$QAST, which can raise IPL to 7 to synchronize with rest of scheduler database.
- If an event flag is requested, it is set by means of SCH$POSTEF.
- Allocates and initializes "replacement" AST control blocks as needed to replenish the UCB AST queue.
- Exits via RSB.

The AST routine gains control in process context, and may respond to the interrupt by:

- Reading/writing device registers (if the process is mapped to the I/O page).
- Interpreting data in the shared buffer, or writing new parameters into the shared data buffer.
- Disconnecting the process from the interrupt by issuing a $CANCEL system service call.

Canceling I/O Requests

The Cancel I/O routine in CONINTERR gains control after $CANCEL request on device is connected to the process and does the following:

- JSBs to IO$CANCELIO to see if there is I/O to cancel.
- If IOC$CANCELIO does not set cancel bit (UCB$V_CANCEL in UCB$W_STS), routine simply returns.
- Otherwise, process-specified cancel I/O routine is called (be means of JSB or CALLS) as appropriate.

  JSB Case:

  ```
  R2      = negated value of the channel index number
  R3      = IRP address
  R4      = PCB address
  R5      = UCB address
  ```

  CALL Case:

  ```
  Ø(AP)   = 4 (argument count)
  4(AP)   = negated value of the channel index number
  8(AP)   = IRP address
  12(AP)  = PCB address
  16(AP)  = UCB address
  ```

Process-supplied cancel I/O routine:

- Gains control at IPL$_QUEAST (6).
- Must preserve all registers except RØ-R3.
- Must not assume channel index number is valid unless it checks UCB$M_BSY in UCB$W_STS to see that process is still connected to the device.
- Places value in RØ and R1 at time of exit into IOSB associated with the connect-to-interrupt $QIO request.
- Must exit at IPL level 6 (IPL$_QUEAST).
- Returns to CONINTERR be means of RET (CALL case) or RSB (JSB case).

Cancel completes as follows:

- UCB is restored to its original state.
- SPTs reserved for process are deallocated.
- Unused AST control blocks are deallocated.
- AST quota is returned to the process.
- $QIO request is completed (if one is outstanding) by invoking REQCOM (determined by checking UCB$V_BSY bit in UCB$W_STS).

digital

EY-2278E-MJ-0001
Printed in U.S.A.