EDUCATIONAL SERVICES

VAX/VMS
Training

VAX/VMS
Device Driver
I/O Sequence

digital

EY-2278E-MC-0001

I/O SEQUENCE

Prepared by Educational Services
of
Digital Equipment Corporation

First Edition, October 1984

Printed in U.S.A.

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

CONTENTS

FIGURES

INTRODUCTION

There are various routines involved in an I/O operation, including system and driver routines. An I/O request is completed through the cooperation and interaction of these various routines.

A driver consists of a collection of separate routines, each of which serves a specific purpose. This module examines the flow of control among the various routines in a typical I/O operation, and describes the sequence of events that takes place in each routine.

OBJECTIVES

Upon completion of this module, you will be able to list the major steps in a typical I/O operation and state the sequence of events for each of the following major steps:

- System Service Dispatching
- $QIO Processor
- FDT Routines
- Driver Start I/O Routine
- Driver Interrupt Service Routine
- I/O Posting
- Driver Timeout Routine
- Driver Cancel I/O Routine
- Driver Controller and Unit Initialization Routines

RESOURCE

1. Guide to Writing a Device Driver for VAX/VMS

TOPICS

- Interrupt Priority Levels (IPL)
- I/O Sequence Overview
- I/O Sequence Specifics
  - FDT Routines
  - Start I/O Routines
  - Interrupt Dispatching and Service Routines
  - I/O Posting
  - AST Interrupts
- Time Out Routines
- Cancel I/O Routines
- Initialization Routines

PRIORITY AND INTERRUPT PRIORITY LEVEL (IPL)

Priority and IPL are separate concepts. A process executing with any process priority can be interrupted by an interrupt condition at any IPL (1-31). Process priority is primarily a software tool used for scheduling, while IPL is mainly a hardware tool used to block out interrupts.

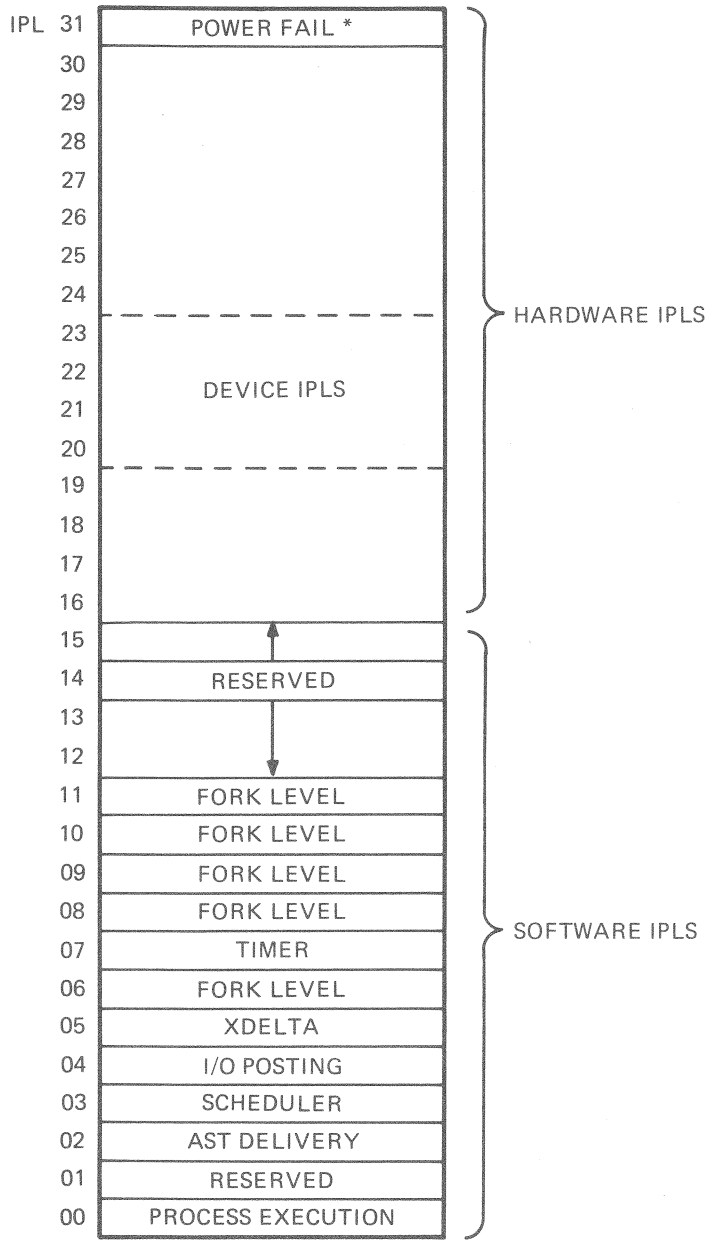
Priority

- Refers to a process priority.
- There are 32 levels of process priority.
- A process with a higher process priority is scheduled before a process with a lower priority.

Interrupt Priority Level (IPL)

- There are 32 levels of IPL (see Figure 3-1).
- An interrupt condition at an IPL lower than the current processor IPL will not interrupt the processor until the processor IPL drops lower than that of the interrupt condition.
- The higher IPLs (16-31) are reserved for hardware (for example), line printer IPL is 20).
- The lower IPLs (1-15) are reserved for software (for example, timer IPL is 7).
- IPL 0 is for normal process execution and is not considered to be an interrupt level.
- IPLs 20-23 are used for device interrupts.
- IPLs 6, and 8-11 are used for fork IPL.

There is no relationship between device IPL and fork IPL. However, devices competing for the same resources (data paths and mapping registers) must use the same fork IPL. Most drivers use fork IPL 8; the mailbox driver uses fork IPL 11. The relationship between device IPL, and UNIBUS BR level discussed in the I/O Architecture module is (Device IPL) = (BR level) + 16.



* POWERFAIL INTERRUPTS OCCUR AT IPL 30, YET IPL 31 IS CALLED POWERFAIL, AND IS USED TO BLOCK OUT ALL INTERRUPTS.

TK-4833

Figure 3-1 Interrupt Priority Levels

I/O SEQUENCE OVERVIEW

All user I/O requests in VAX/VMS are either issued as $QIO system service requests, or are converted into $QIO system service requests by RMS.

A $QIO system service request results in a CALL to the **System Service vector** (SYS$QIO) which dispatches the request to the **$QIO Processor** (EXE$QIO). The $QIO processor performs an initial check on the validity of the I/O request, creates an IRP, and invokes the **FDT Routine(s)**. The FDT Routine(s) performs some function/device-dependent work and initiates the **Driver Start I/O Routine** if the driver is not busy. If the unit is busy, the IRP is queued. Up to this point, all processing is carried out in the context of the process that requested the I/O.

The driver Start I/O Routine initiates the device hardware to perform the requested I/O, then waits for a device interrupt to occur by issuing the WFIKPCH macro. When the device interrupts, the **Driver Interrupt Service Routine** determines which device unit interrupted, and resumes the driver fork process that is waiting for that particular interrupt. The resumed driver fork process executes at hardware device IPL, which blocks out many other operations. The driver fork process usually "forks" at this point to lower IPL. Eventually, the "fork" **dispatcher** resumes the "forked" driver process. The driver process then performs any interrupt specific processing, such as clearing an attention condition, and finally invokes the **I/O Postprocessing Routine** (by issuing the REQCOM macro). The I/O Postprocessing Routine returns I/O data, I/O status and deallocates the IRP. Figure 3-4 illustrates a typical I/O sequence. It does not include any abnormal condition handling such as time-out, powerfail or cancel I/O and depicts only the major steps involved in a typical, successful I/O operation.

It is possible that an ACP or XQP is involved during an I/O request related to file or network processing. The FDT routines check the function code and invoke the corresponding ACP/XQP if one is required. An ACP/XQP usually handles virtual to logical conversion for file I/O, and handles protocols for network I/O. After completing its part of the work, an ACP/XQP may invoke the driver to finish the I/O processing. More detailed information on ACP/XQPs is presented in the Related Topics module.

I/O REQUEST PACKET QUEUING

Issuing a $QIO system service request results in a CALL to a system service vector SYS$QIO (see Figure 3-6). The vector contains a register save mask, a CHMK #QIO instruction, and a RET instruction. Execution of the CHMK instruction causes an exception to occur. The exception is vectored through the System Control Block to the change mode dispatcher. The change mode dispatcher picks up the exception code and decides whether or not it is legitimate. It checks that the argument list is the correct length for a $QIO, and that the argument list is read-accessible for the access mode of the user process. The change mode dispatcher then dispatches control to the $QIO Processor (EXE$QIO; see Figure 3-7).

Note that if a read or write virtual I/O request is issued, the function code is converted to the corresponding read or write logical I/O request by EXE$QIO.

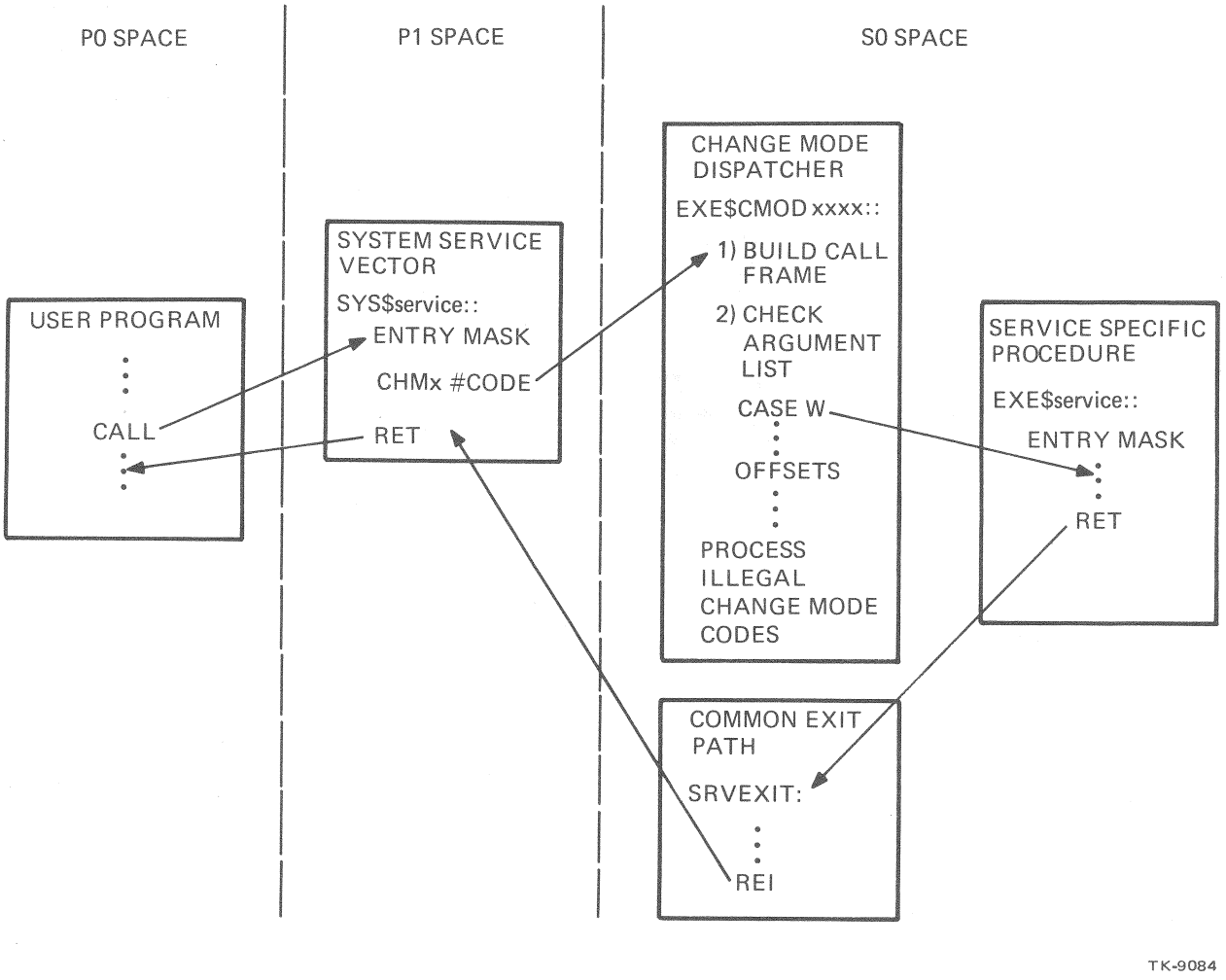


Figure 3-2 Control Flow of System Services that Change Mode

I/O PROCESSING

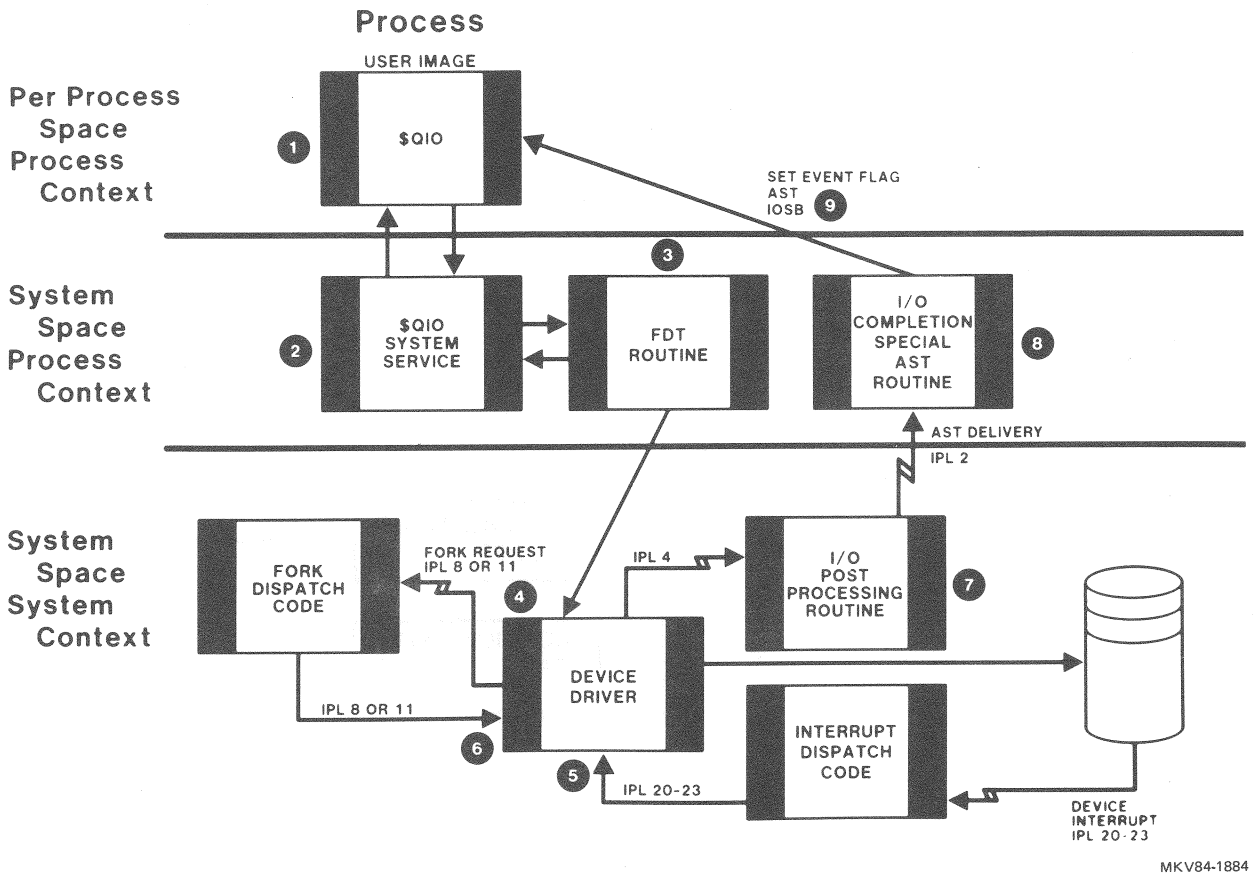


Figure 3-3 Input/Output Flow (Full)

Sequence		
1	User request	Process Context
2	$QIO System Service code	
3	FDT routines	System Context
4	START I/O routine (part I)	
5	Interrupt Service routine	
6	START I/O routine (part II)	
7	I/O post-processing	
8	Special K Ast	Process Context
9	User Ast, if specified	

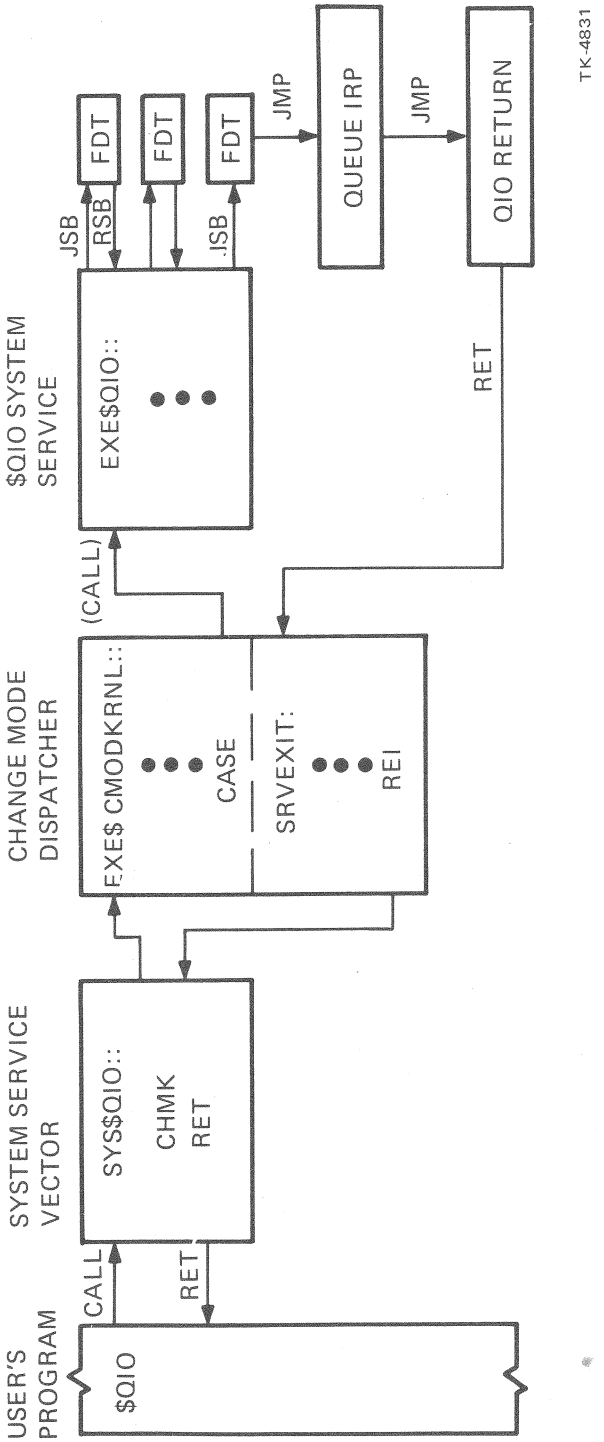


Figure 3-4 Basic Sequence of Events in Dispatching to the $QIO System Service

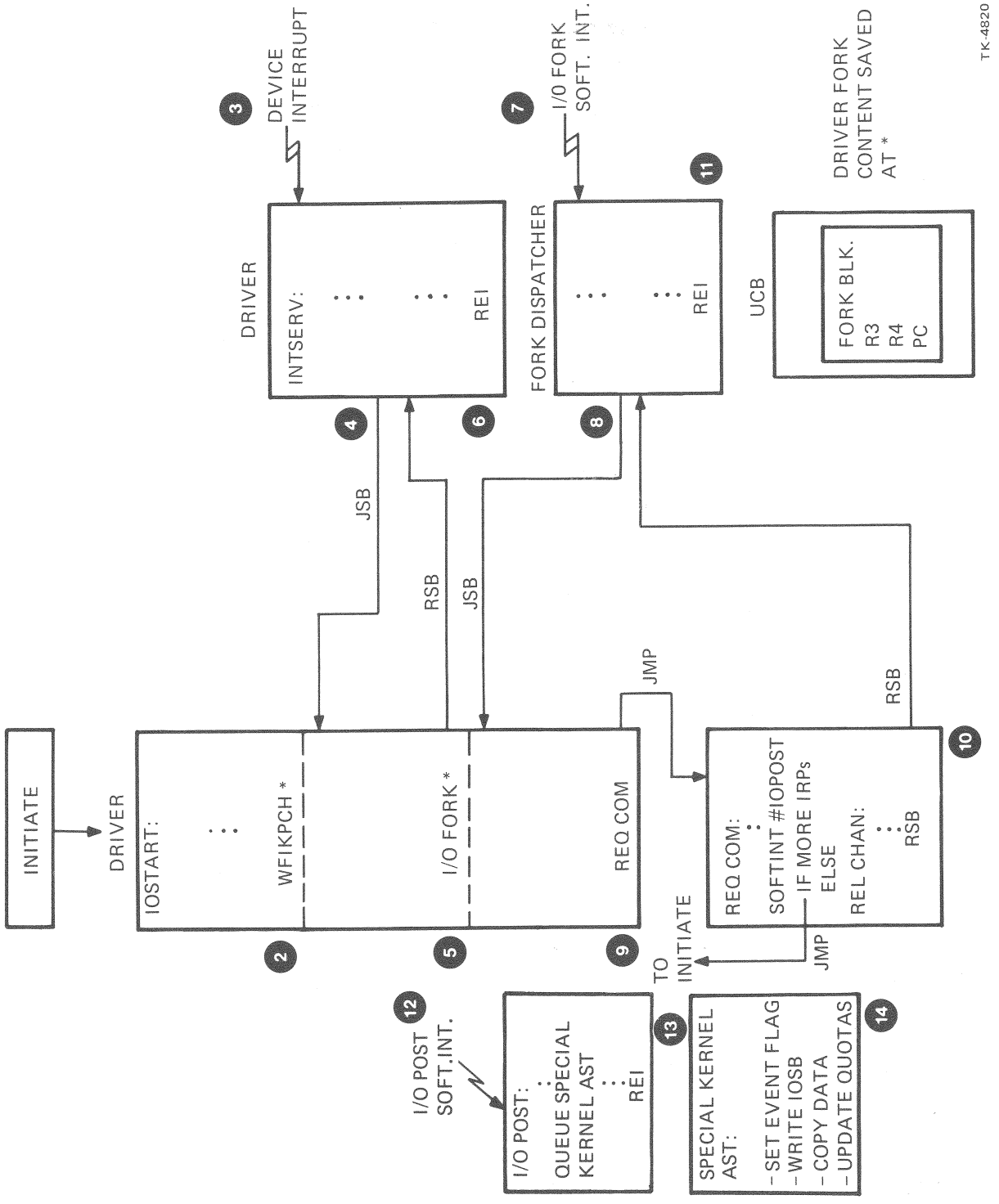


Figure 3-5 Processing $QIO Requests

Notes on Figure 3-5

1. Start transfer
2. Wait for Interrupt
3. Interrupt generated
4. JSB to driver code
5. Create fork process RSB to Interrupt server
6. Dismiss Interrupt
7. Interrupt at Fork IPL (Fork Dispatcher)
8. JSB to driver code
9. JMP to Request Complete routine
10. RSB back to Fork Dispatcher
11. Dismiss Interrupt
12. Interrupt at I/O Post IPL
13. Dismiss Interrupt
14. Special K Ast

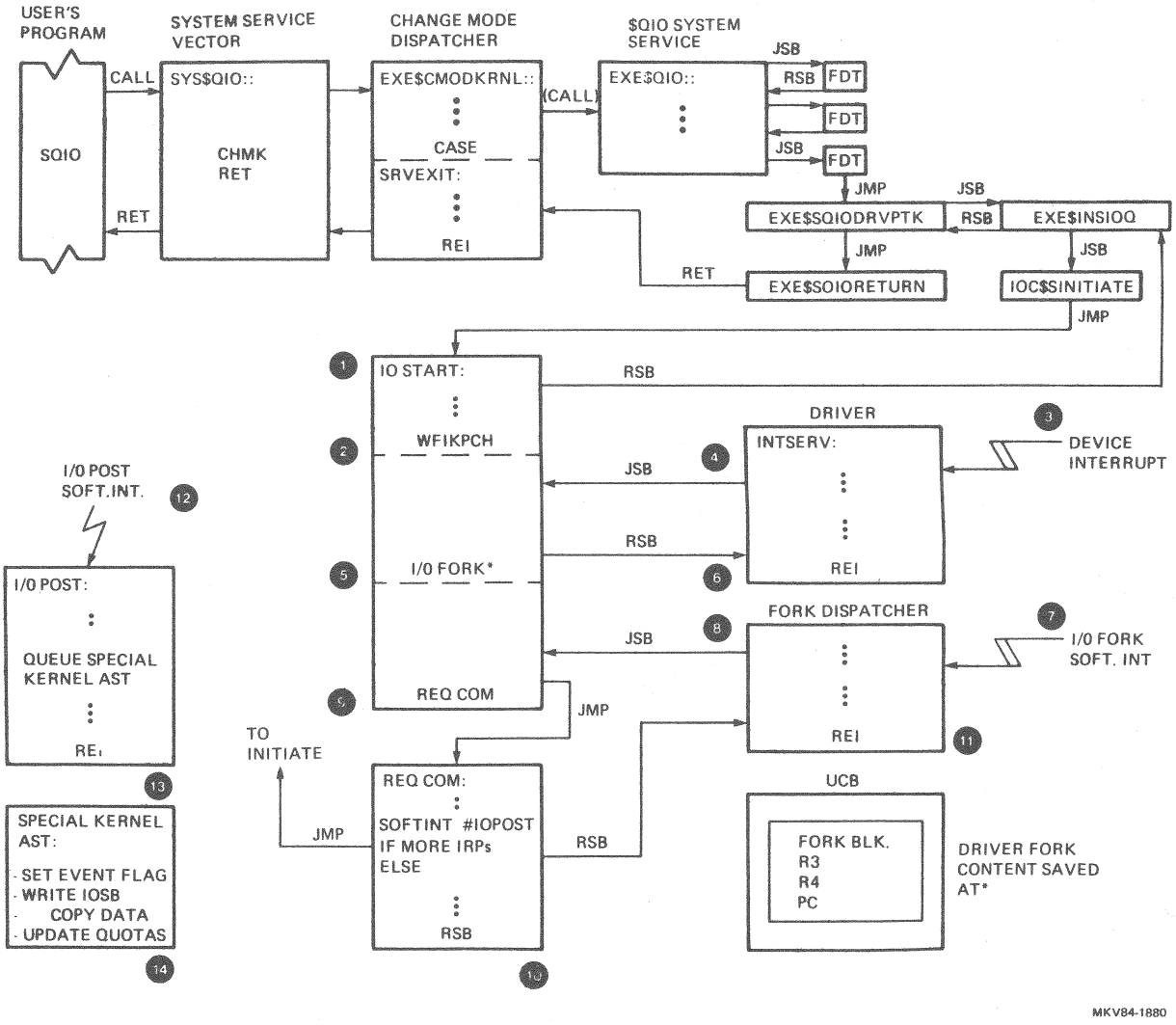


Figure 3-6 Detailed Sequence of Events in Dispatching to the $QIO System Service

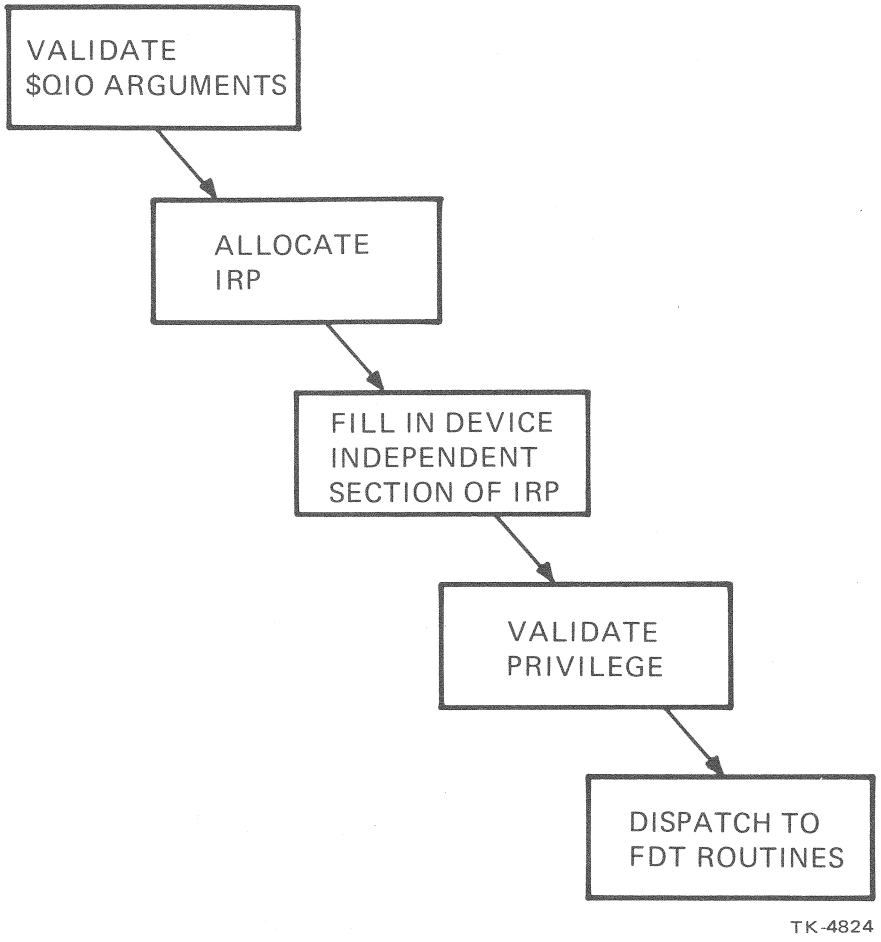


Figure 3-7 The $QIO Processor (EXE$QIO)

This routine is responsible for the device-independent work involved in processing an I/O request.

FDT ROUTINES

The FDT Routines are responsible for filling the device/function-dependent section of the I/O Request Packet (storing the P1-P6 parameters). Typically, they call an executive subroutine to check the read/write accessibility of the user's buffer. For a direct I/O request, they call a subroutine to lock the buffer pages in memory; for a buffered request, they allocate a buffer in system nonpaged memory (after verifying that the user has sufficient quota). Figure 3-8 illustrates the kinds of operations performed by FDT routines.

FDT routines are invoked (and run) in process context at IPL 2. Typically, they do not raise IPL, and must not lower it below 2. Executing at IPL 2 blocks delivery of ASTs to the process while allowing the FDT routine code to be pageable. Blocking an AST that might cause process deletion is required while the process is holding the only pointer to the allocated I/O request packet.

FDT routine is terminated by:

- An RSB (which results in another FDT routine being invoked).
- Jumping to a specific executive routine to perform a selected function.

Function	Executive Routine Name
Abort the I/O	EXE$ABORTIO
Finish I/O; return full IOSB	EXE$FINISHIO
Finish I/O; return first half of IOSB	EXE$FINISHIOC
Queue IRP to driver	EXE$QIODRVPKT (also invokes EXE$INSIOQ)
Queue IRP to ACP	EXE$QIOACPPKT
Queue IRP to XQP	EXE$QXQPPKT
Queue IRP to alternate; start I/O entry point in driver	EXE$ALTQUEPKT

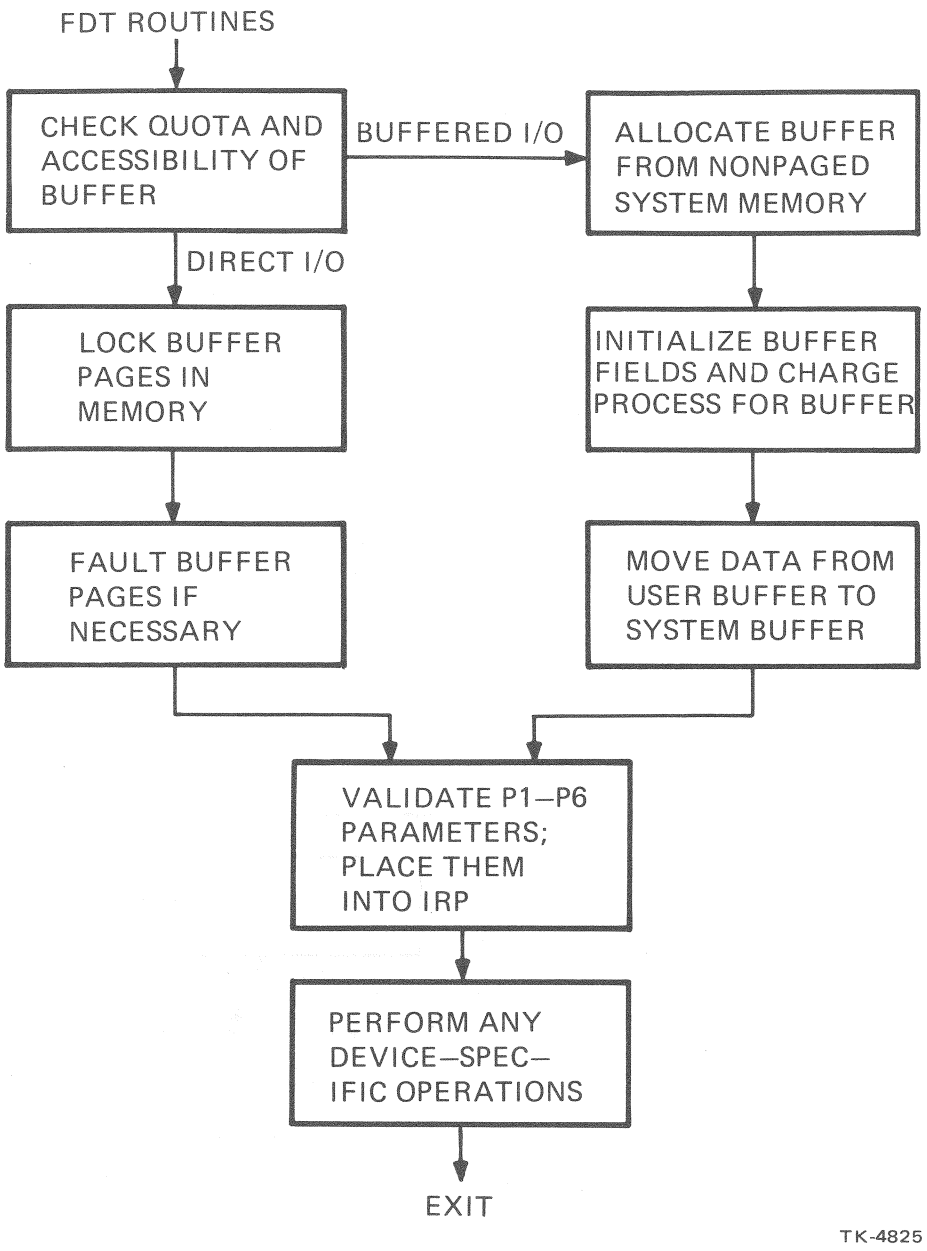


Figure 3-8 FDT Routines

The work done by FDT routines is device/function-dependent. This figure illustrates the typical operations performed for FDT routines supporting drivers using buffered and direct I/O.

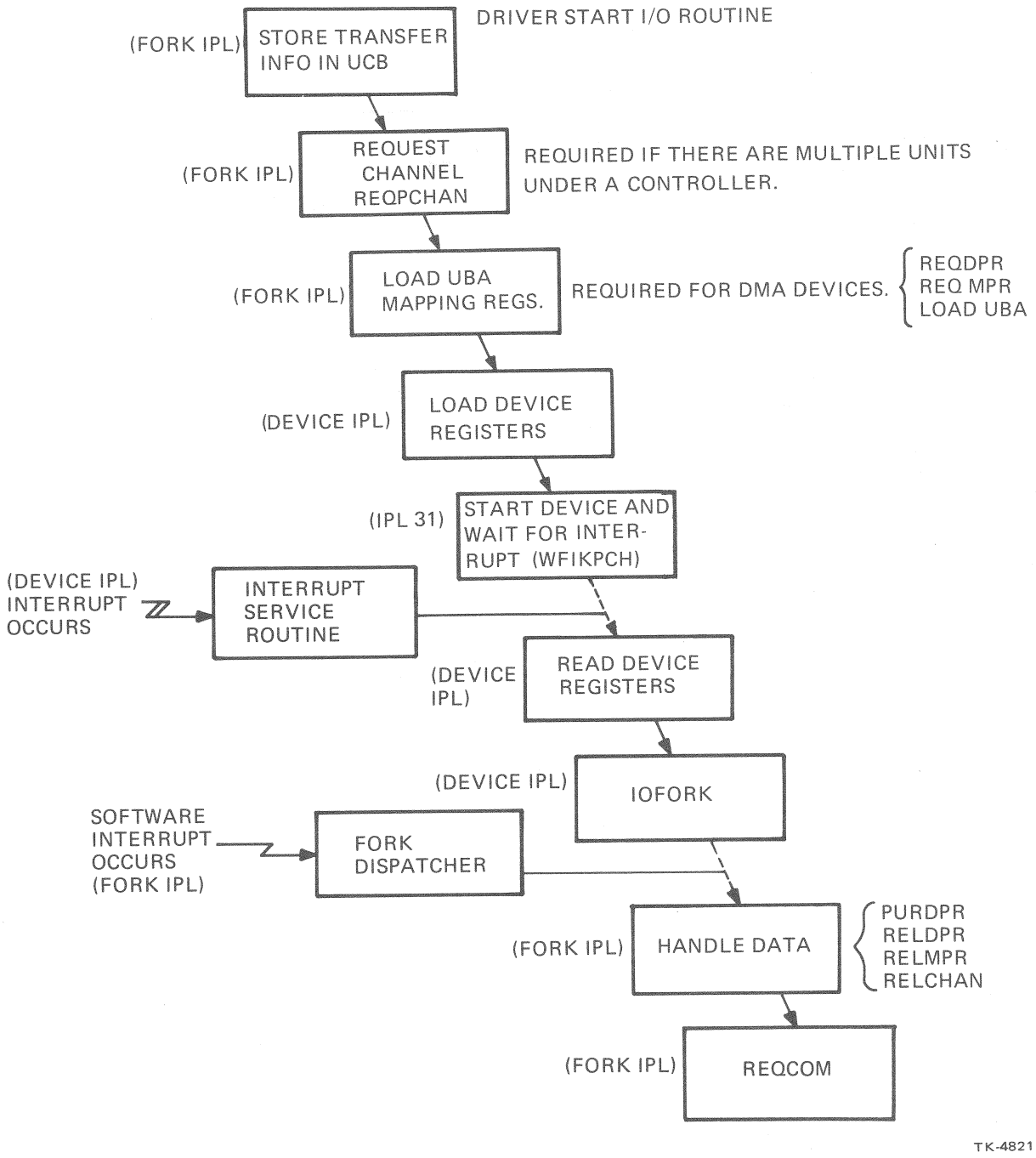


Figure 3-9 General Flow of the Start I/O Routine of a Driver

DRIVER START I/O ROUTINES

The I/O operation is started by the appropriate device driver code executing at elevated IPL (device fork level). Although driver code is device specific, a number of device independent support routines may be evoked to perform common functions (request channel, wait for interrupt; see Figure 3-9).

Starting the I/O operation involves:

- Initializing certain fields and inserting function dependent parameters in the UCB (from fields in the IRP).
- Requesting the device controller channel where appropriate [REQPCHAN macro].
- Requesting UBA mapping registers, and loading those registers [REQDPR, REQMPR, and LOADUBA macros for UNIBUS; REQMPR and LOADMBA for MASSBUS].
- Setting device registers (for example, byte count, starting address).
- Starting the device (probably by setting a bit in a device register).
- Returning to the requesting process to wait for interrupt [WFIKPCH macro].

The following main data blocks are used:

- I/O Request Packet (IRP), which contains the device and function dependent parameters of the I/O request.
- Unit Control Block (UCB), used to store device and function dependent parameters for subsequent use by the driver in loading device registers.
- Driver Dispatch Table (DDT), to locate the start I/O entry point of the driver.
- Channel Request Block (CRB) and Interrupt Data Block (IDB), used in the arbitration of I/O requests for multiunit controllers.

Returning to the Requesting Process

Before setting the 'go' bit for the device, the driver process raises IPL to 31 in order to block powerfail interrupts (which occur at IPL 30). It then checks to see if a powerfailure occurred. If not, the driver sets the appropriate bit in the CSR to start the hardware, stores the driver context in the fork block, lowers IPL, and returns to the requesting process to wait for an interrupt. If a powerfailure occurred, the driver may either abort the I/O operation, or reload the device registers (which may have been destroyed by the powerfailure) and retry the I/O operation.

Continuation After Interrupt

Sometime later, an interrupt occurs. The system interrupt dispatcher and the driver interrupt service routine find the device unit that expects the interrupt (details can be found in the next section), and restore its driver fork process context. At this point, the driver fork process is executing at device IPL. Any device registers to be read should be read at this point and saved somewhere in the UCB. After reading device registers, the driver fork process issues the IOFORK macro to lower IPL. IOFORK queues the driver fork process to the appropriate fork queue. Eventually, the fork dispatcher gains control, and dequeues the driver process from the fork queue. The driver fork process then does what is necessary with the data that was read earlier, returns allocated resources to the system (RELCHAN, RELMPR, RELDPR macros), and invokes system macro REQCOM to start I/O postprocessing.

INTERRUPT DISPATCHING

Four sequences comprise interrupt dispatching:

1. Locating the Adapter Control Block representing the bus on which the interrupting device exists.

2. Vectoring to the correct interrupt service routine.

3. Finding the device unit requiring attention.

4. Resuming the driver process waiting for the interrupt.

Interrupt dispatching is carried out at an IPL corresponding to the UNIBUS device Bus Request level (BR level):

BR4 = IPL 20
BR5 = IPL 21
BR6 = IPL 22
BR7 = IPL 23

The five primary data structures providing the basic dispatching information are:

- **System Control Block (SCB)**, which contains entries for each possible exception and interrupt. It is used by the CPU microcode to locate the appropriate ADP.

- **Adapter Control Blocks (ADPs)**, which contain code to locate the CRB associated with the interrupt.

- **Channel Request Blocks (CRBs)**, which contain code to activate the interrupt service routine appropriate for each device type. There is one CRB for each controller. A CRB contains multiple slots to handle multiple-port (multivector) devices.

- **Interrupt Data Blocks (IDBs)**, which provide the interrupt service routine with the address of the device control/status register (CSR), and the addresses of the UCBs. The IDB also points to the UCB corresponding to the unit which currently owns the controller, if any. There is one IDB for each controller.

- **Unit Control Blocks (UCBs)**, which provide the device unit status, and the driver fork process context (fork blocks). There is one UCB for each device unit.

For a UNIBUS device interrupting through the UBA, the interrupt is vectored through one of four SCB vectors corresponding to the four BR levels (4, 5, 6, 7) on the UNIBUS. Each SCB vector points to interrupt service routine (located in the ADP), which saves general registers RØ-R5 and reads the corresponding Bus Request Receive Vector Register (BRRVR) to obtain the UNIBUS vector address. This UNIBUS vector address is used as an index to the vector table (VECTAB in the ADP) to find the CRB associated with the interrupting device. (The pointer to the CRB was placed in VECTAB by the driver loading procedure, SYSGEN, when the unit was connected.)

On the VAX-11/75Ø and VAX-11/73Ø, there are directly vectored interrupts. Extra entries in the SCB (which is larger) point directly to the CRB.

For each vector that a device can specify, the CRB contains a JSB instruction, followed by the address of an Interrupt Data Block (IDB). The JSB instruction pushes the address of a pointer to the IDB onto the stack, and transfers control to the driver interrupt service routine.

Driver Interrupt Service Routine

The driver interrupt service routine uses the IDB to obtain the address of the controller's control/status register (CSR) and the owner unit UCB address (or the address of the appropriate UCB, if there is no owner).

For a single unit controller, the UCB status word is examined to see if it is expecting an interrupt. If yes, the unit's driver process context is restored from the fork block, and the driver process continues from the point where it left off (waiting for an interrupt). If no, the driver's interrupt service routine decides how to handle (dismiss) the interrupt for UNIBUS devices.

Interrupt Service routines for multiunit controllers are discussed in the Required Driver Routines module.

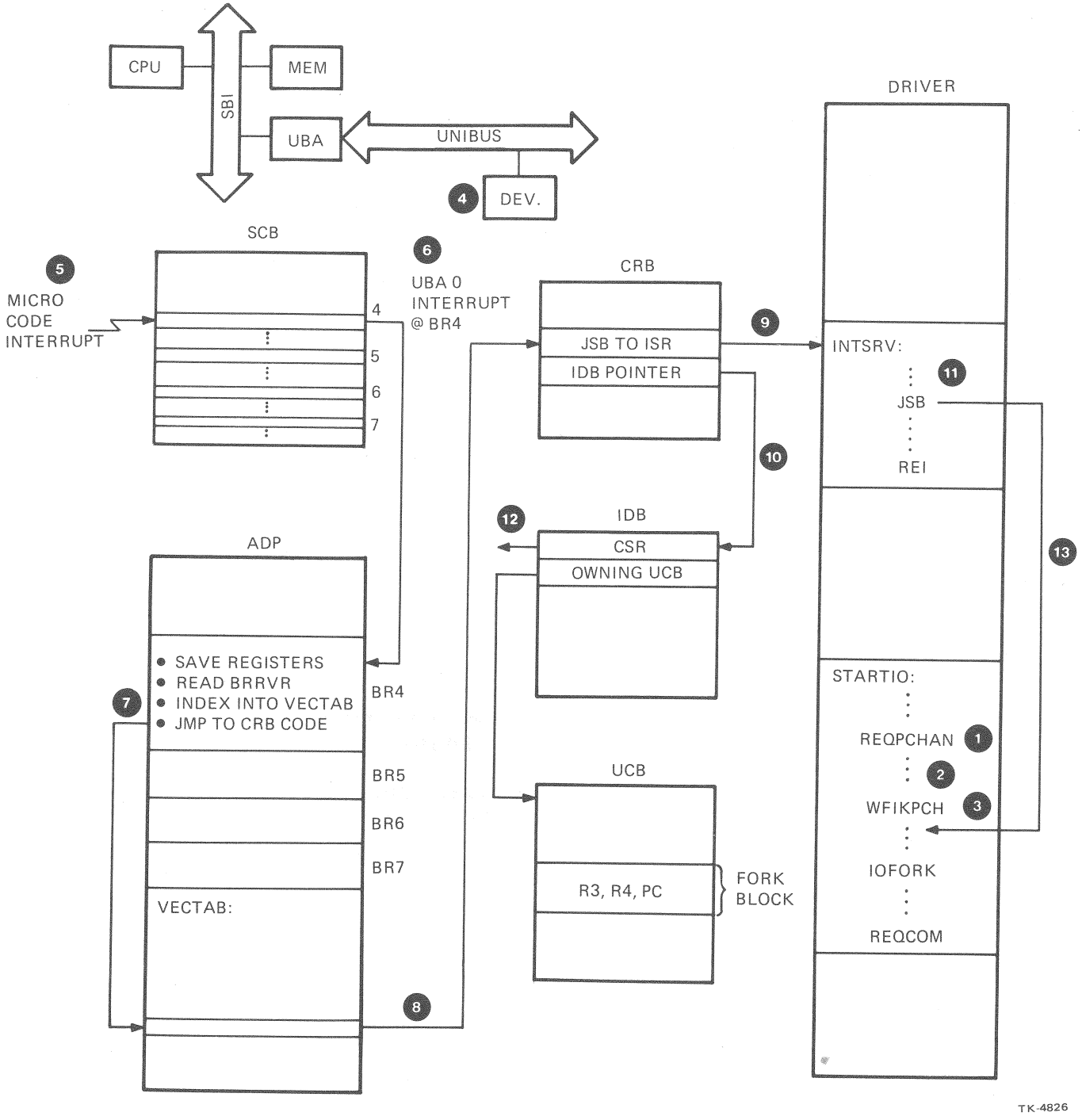


Figure 3-10 UNIBUS Device Indirect Interrupt Dispatching for VAX-11/780

(The order of events is indicated by number.)

UNIBUS Device Interrupt Dispatching

1. REQPCHAN fills in 'Owning UCB' field of IDB.
2. Device is started.
3. WFIKPCH fills in Fork Block in UCB.
4. Device interrupts UBA; UBA interrupts VAX processor.
5. Microcode indexes into SCB to entry for this UBA at this IPL (BR level).
6. Boot procedure loaded SCB entries with pointers to ADP code.
7. ADP code saves registers and indexes into VECTAB area based on interrupt vector specified by device.
8. SYSGEN pointed VECTAB entry to code in CRB when driver was loaded.
9. SYSGEN pointed JSB in CRB to interrupt service routine when driver was loaded.
10. SYSGEN also pointed next longword of CRB to IDB.
11. Interrupt service routine executes POPL to move pointer to IDB.
12. SYSGEN filled pointer in IDB to CSR with system virtual addresses mapped to UNIBUS I/O page.
13. Interrupt service routine locates fork block in UCB and resumes fork process executing driver code (at device IPL).

INTERRUPT DISPATCHING DIFFERENCES

There is a slight difference in UNIBUS interrupt dispatching on the 750 and 730 which is transparent to the driver. When a UNIBUS device generates an interrupt on the 750 or 730, the interrupt is vectored through the SCB, and control is immediately transferred to location CRB$L INTD in the appropriate CRB, after which interrupt dispatching proceeds as in the case of the 780. Figure 3-11 illustrates 730/750/780 interrupt dispatching differences.

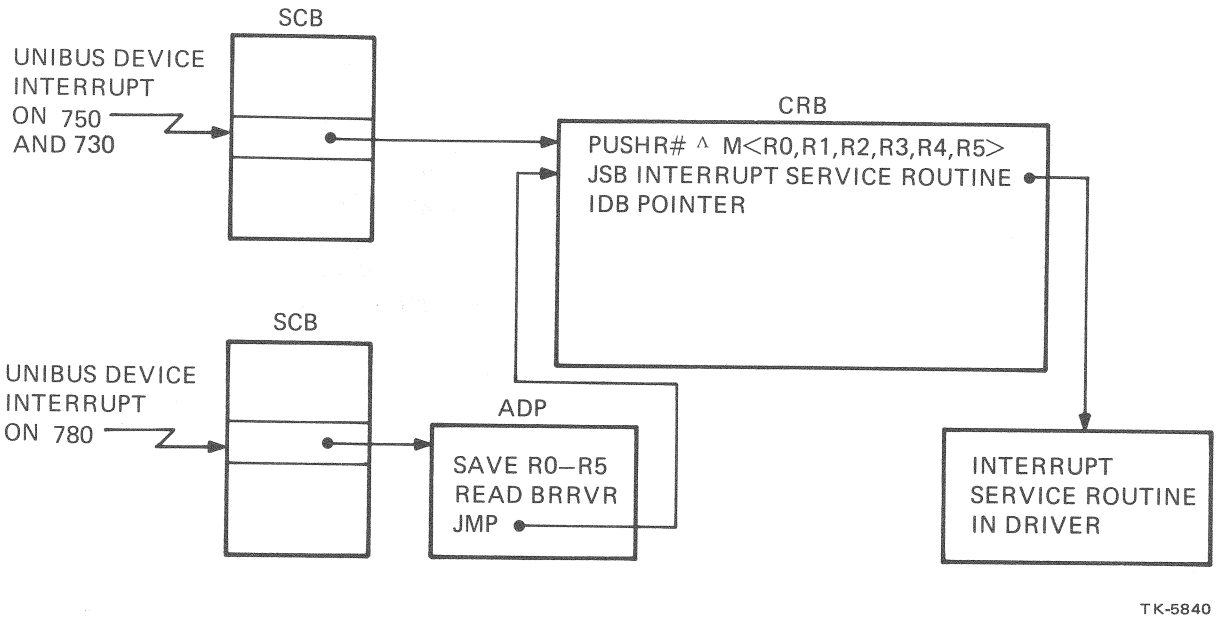


Figure 3-11 730/750/780 Interrupt Dispatching Differences

The 750 and 730 have no interrupt service routines in the ADP, and the UBA itself never generates an interrupt. The size of the SCB is increased to accommodate all of the UNIBUS interrupt vector locations. The address for all interrupt vectors is given by:

SCBB (SCB base) + 200 (hex) + device vector (octal)

When a unit is CONNECTED (via SYSGEN), the appropriate vector field(s) is(are) initialized in the SCB to point to the CRB for the device controller.

MASSBUS Device Interrupt Dispatching

Interrupt dispatching is handled differently for MASSBUS devices. When the system is booted, CRBs and IDBs are created, and entries are made in the SCB to transfer control to locations in the CRB for each available MASSBUS adapter. The instructions in the MBA CRB are a PUSHR for R2-R5, and a JSB to the MASSBUS adapter interrupt service routine MBA$INT (part of the executive, not located in the ADP for the MASSBUS adapter).

The MBA$INT routine responds to the interrupt and, for solicited interrupts for single-unit controllers, transfers control directly to the instruction following the WFIKPCH macro (without calling an intermediate driver-written interrupt service routine). If a single-unit controller is not expecting an interrupt, the unexpected interrupt routine is called. For multidevice controllers, control is transferred to the interrupt service routine of the driver (which must determine if the interrupt is expected or unexpected). Figure 3-12 illustrates the general control flow(s) in MASSBUS interrupt dispatching. Figure 3-13 contains a more detailed outline of the logic flow in the MASSBUS adapter interrupt service routine MBA$INT.

- Single-device controllers do not have an interrupt service routine in their drivers (since MBA$INT never transfers control to an interrupt service routine for single-device controllers).

- Single-device controllers may, however, have an unsolicited interrupt service routine. This routine exits with an RSB, **not** by restoring R2-R5 and issuing an REI (which is what MBA$INT does).

- Interrupt service routines, for multidevice controllers must determine whether the interrupt is expected or unexpected (and which unit requires attention). Control is never transferred to an unsolicited interrupt service routine (for a multidevice controller) by MBA$INT. In addition, the interrupt service routines must restore R2-R5 and issue an REI following the JSB instruction that transfers control back to the driver at the instruction following WFIKPCH.

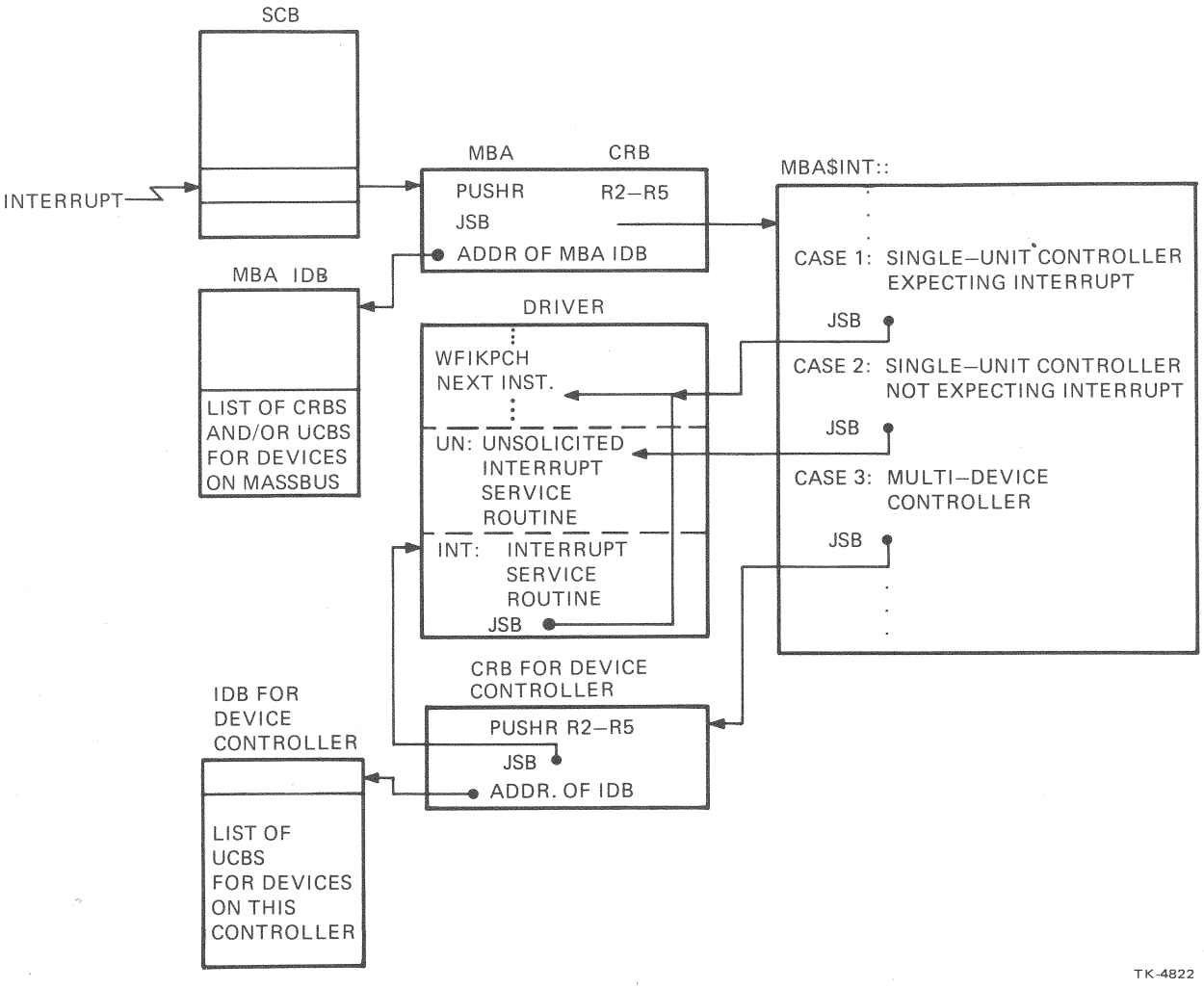


Figure 3-12 General Control Flowpaths for Processing MASSBUS Interrupts

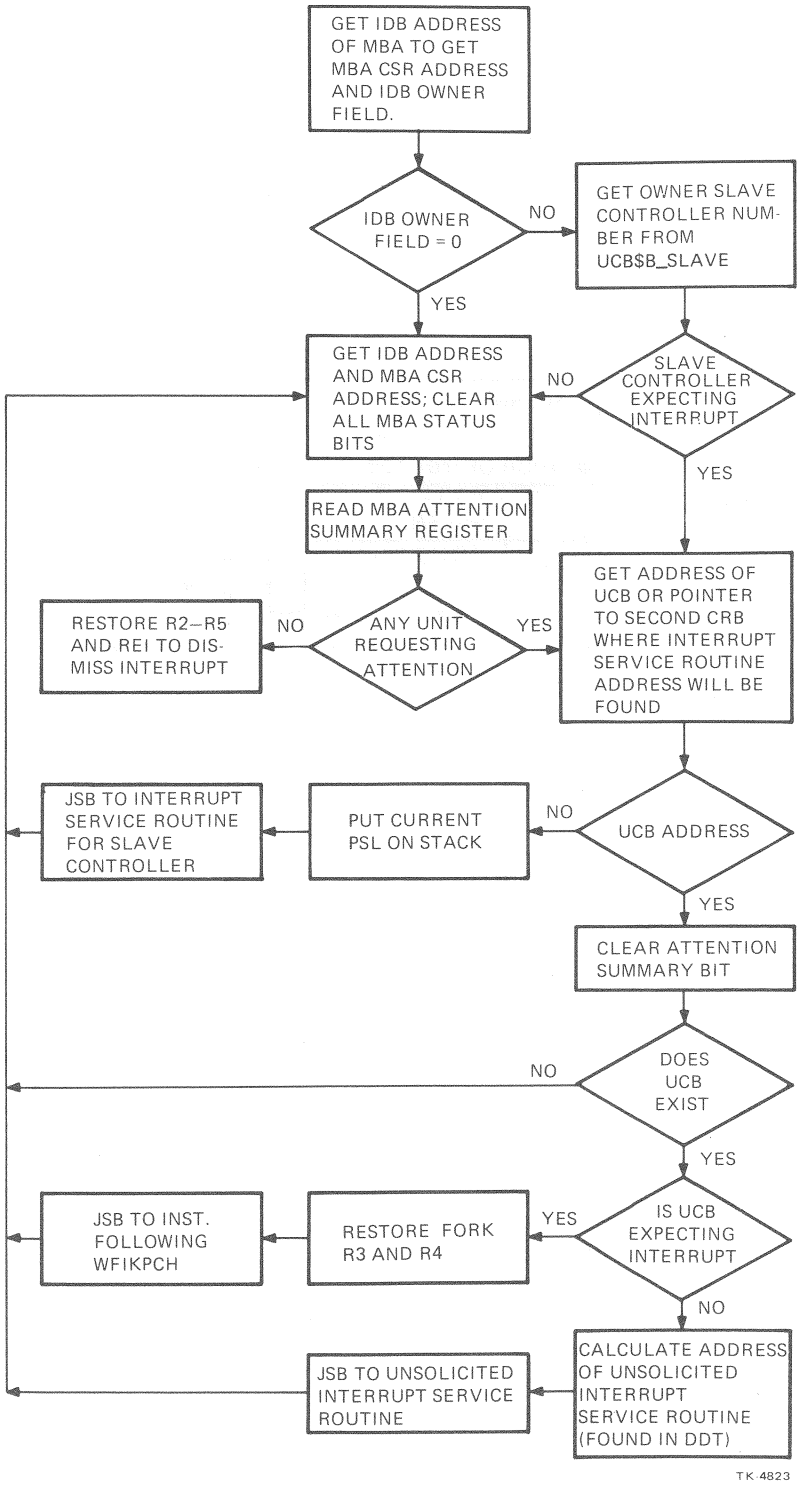


Figure 3-13 Logic Flow in MASSBUS Adapter Interrupt Service Routine MBA$INT

- Drivers wanting to use RØ and/or R1 must save and restore them since they are not saved in either PUSHR instruction (in either CRB).

- Interrupt service routines for multidevice controllers are responsible for clearing the MBA attention summary bit for the interrupting device (this bit is always cleared for single-unit controllers).

- MBA$INT decides whether an entry in the MBA IDB is a UCB address (single-unit controller), or a pointer into a CRB (multidevice controller) by checking the low order bit of the entry in the MBA IDB for the controller. If the bit is set, the entry is for a multidevice controller; if the bit is clear, the entry represents the UCB address for the device on a single-device controller. UCBs, like CRBs, are always longword aligned (that is, the low order bit is clear). When a CRB is created for a multidevice controller, and its address is stored in the MBA IDB, the address is increased by 1 to set the low-order bit. Control is actually transferred to the PUSHR instruction in the multidevice controller CRB via a JSB -(R5) instruction (where R5 contains the MBA IDB entry), so that the low order bit is cleared before control is transferred.

- MBA$INT always checks the attention summary register when an interrupt service routine returns to determine whether another device on the MASSBUS requested an interrupt while the MASSBUS owner was transferring data, or while the current interrupt was being processed. Data transfer functions block the interrupts from nontransfer functions until the data transfer completes.

- The PSL is pushed on the stack before issuing a JSB to an interrupt service routine so that when the interrupt service routine issues an REI, a valid PSL will be present. Control will return (after the REI in the driver) to the instruction following the JSB in the interrupt service routine (in MBA$INT). MBA$INT itself issues an REI (eventually) to dismiss the initial interrupt. Note that several REIs can be (and in fact, are) issued to handle one interrupt. For each REI, there must be a valid PC and PSL on the stack.

- MBA$INT may be found in module MBAINTDSP of the executive source listings.

I/O POSTING

After issuing the REQCOM macro, the driver no longer has any control of the I/O operation. All I/O postprocessing is handled by the executive, as seen in Figures 3-14 through 3-16.

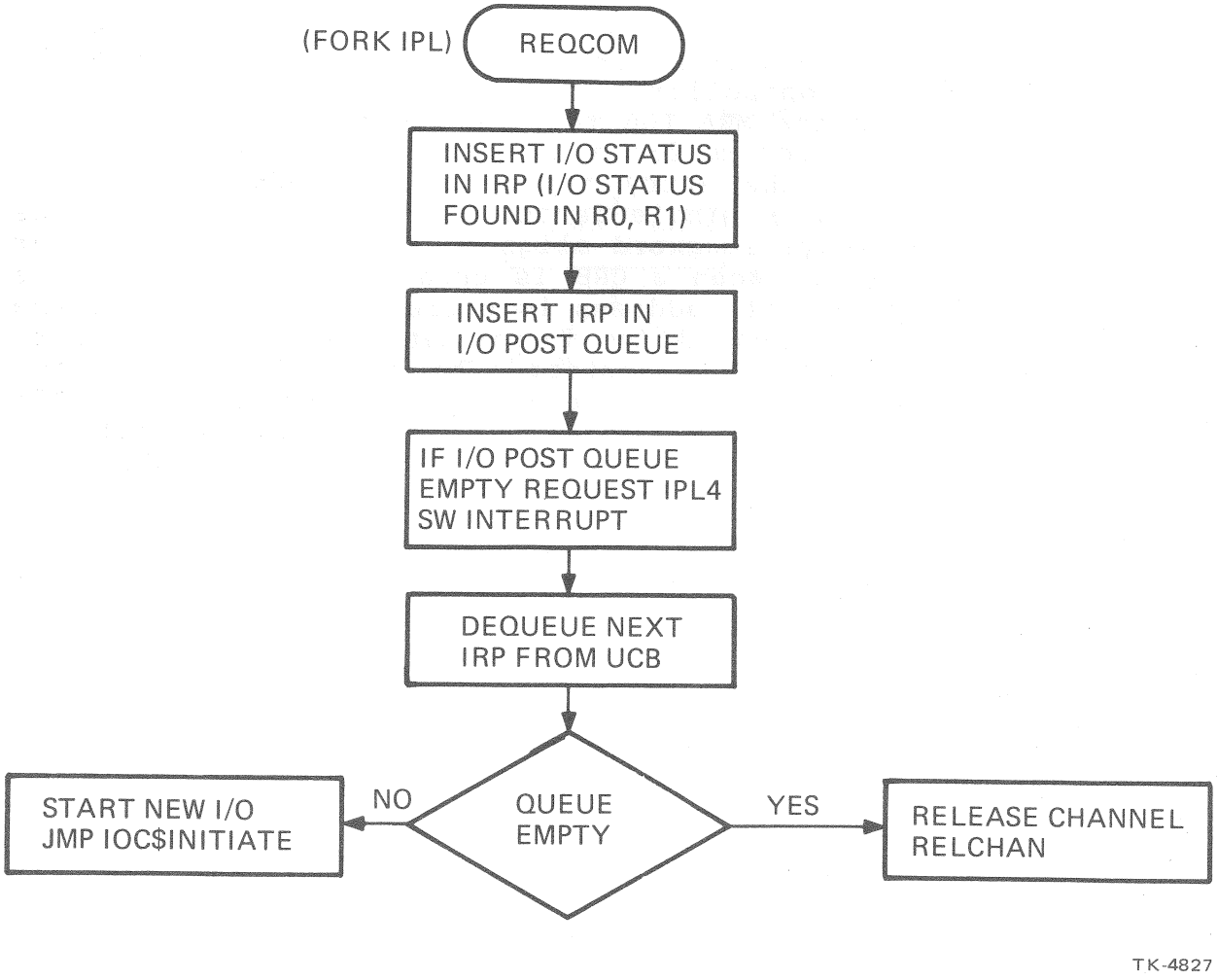
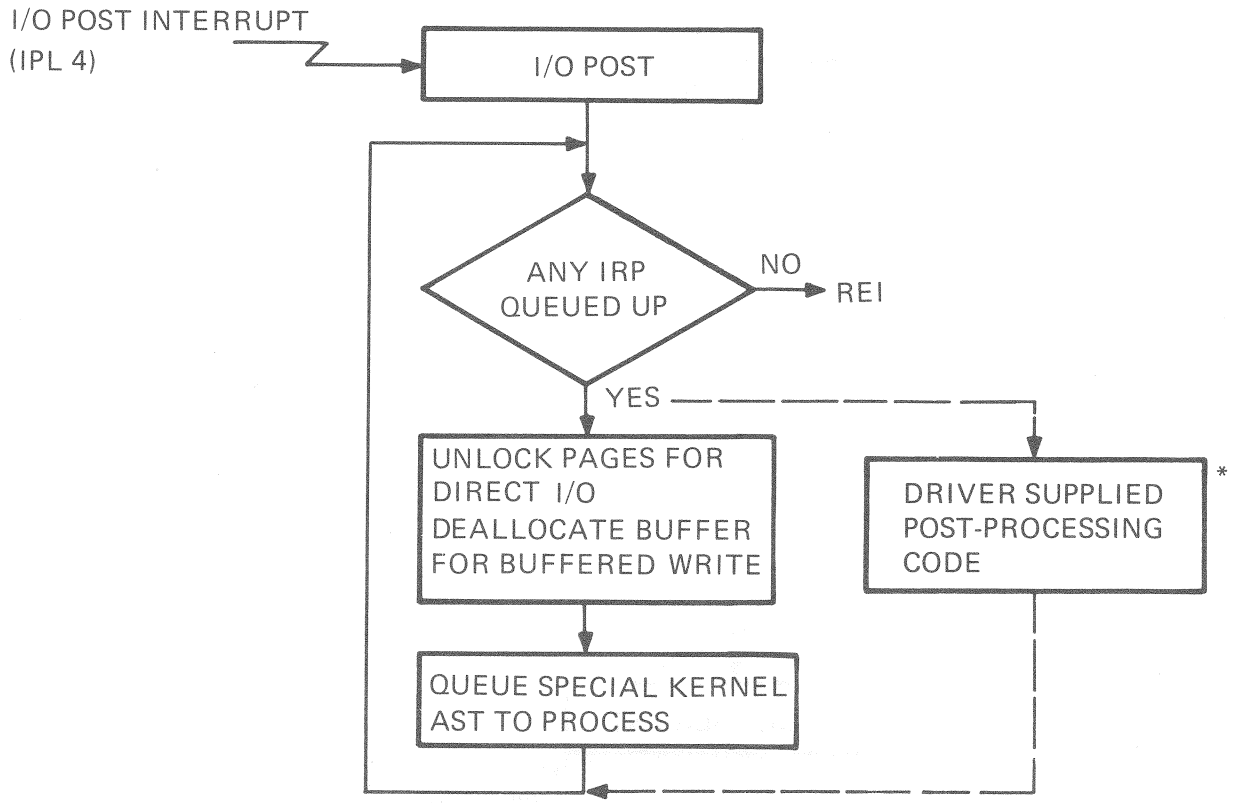


Figure 3-14 I/O Posting

The last instruction in the driver START I/O Routine is REQCOM. It is a system macro that starts I/O Posting.

I/O Post Interrupt

When the processor IPL drops below 4, the I/O Post interrupt is honored. Note that the IRP is used as the special kernel AST control block.



*IF IRP$L—PID HAS HIGH ORDER BIT SET, IT IS INTERPRETED AS AN S0 ADDRESS OF DRIVER SUPPLIED POST PROCESSING ROUTINE.
EXAMPLE: AVOIDS NORMAL POST PROCESSING CODE

TK-4828

Figure 3-15 I/O Post Interrupt

AST Interrupt

When the Processor IPL drops below 2, the AST interrupt occurs. The IRP is used again as an AST control block, if the user requested an AST.

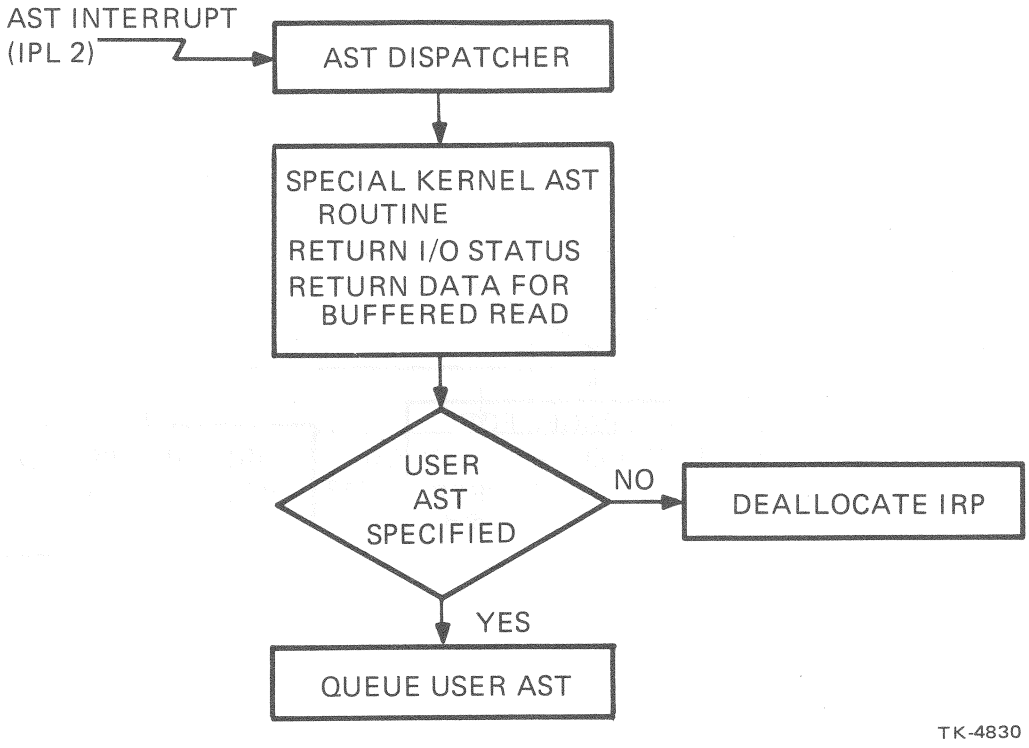


Figure 3-16 AST Interrupt

REQCOM is entered at fork IPL. It stores the I/O status in the request's IRP, places it in the I/O postprocessing queue, and requests an I/O post interrupt. I/O postprocessing may occur much later because the I/O post interrupt occurs at IPL = 4. REQCOM goes on (at fork IPL) to examine the unit's I/O queue in an attempt to start the next request, to keep the unit as busy as possible. If the queue is empty, the unit busy bit is cleared, and control is returned to the fork dispatcher. Otherwise, the next request is dequeued and the driver is called at its start I/O entry point. The driver process initiates the specified function, and then returns to the fork dispatcher which dispatches the next queued fork block, if any.

I/O postprocessing occurs after the driver exits and the IPL drops sufficiently to receive the I/O post interrupt.

When I/O Post runs as a result of the IPL 4 software interrupt, IRPs are pulled out of the postprocessing queue and processed until they are finished. I/O Post balances quota counts and checks the IRP status flag word to determine if the request is direct or buffered I/O. For direct I/O, pages that were locked are unlocked. For buffered write operations, the buffer is allocated. For buffered read operations, the data must be transferred from the allocated buffer to the user's buffer in the user's address space.

In any case, the I/O status block must be written into the user's address space. To do this, a special kernel AST is queued to the process that initiated the I/O request. The IRP becomes an AST control block and is placed into the AST queue for the process that requested the I/O. The kernel AST routine address is set up to be a part of the I/O Post code.

The I/O post interrupt service routine then loops back to pick up another IRP to process. If there are no more IRPs to process, I/O Post issues an REI. The user is scheduled sometime later because there is an AST pending.

When the user begins to execute, s/he immediately receives an AST in kernel mode. That AST causes the I/O Post AST routine to run. The routine knows that the AST control block is also an IRP. It pulls the parameters out from the AST Control Block, which are necessary to transfer the I/O status back to the user. Then, it probes the user's address space to make sure that the piece of memory specified on the way in as an I/O status block, or a buffer, is still there, and that it is still accessible to the user.

If both checks succeed, the I/O status is written to the user-specified I/O status block. If the function is a buffered read, the buffer contents are transferred to the user's buffer. If the user requested an AST, the special kernel AST requeues the I/O packet to the AST queue. The user-specified AST is delivered in the access mode of the original $QIO. If no AST was requested, the special kernel AST deallocates the I/O request packet, and returns.

TIMEOUT ROUTINE

Setup

- By issuing WFIKPCH TIMEOUT_ADDR, #SECONDS which:
  - sets UCB$V_TIM (timeout expected) and UCB$V_INT (interrupt expected) in UCB$W_STS.
  - places I/O duetime in UCB for time when timeout should occur (UCB$L_DUETIM).
- Note that granularity of timeout is ±1 second (must specify at least a 2 second timeout).

Entered

- If system time is greater than or equal to I/O duetime.
- At device IPL (with stack in same state as for an interrupt service routine).
- After powerfail (with UCB$V_POWER bit set in UCB$W_STS).

Responsibilities

- Reinitiate I/O operation, if possible (maybe update a retry counter).
- Send error message to operator mailbox after n timeouts, if appropriate.
- Cancel the I/O request, if no recovery possible.
- Clear the UCB$V_POWER bit in UCB$W_STS after powerfailure.

CANCEL I/O ROUTINE

- Receives dispatches (from EXE$CANCEL) through cancel I/O routine address stored in DDT,
  - when channel is deassigned.
  - when device is deallocated.
  - when $CANCEL service is executed.
  - when image exits, or process is deleted.

The cancel I/O routine typically invokes a system subroutine IOC$CANCELIO which verifies that the:

- Unit is busy.
- Current IRP is for "this" process.
- Current IRP is for "this" channel.

If all conditions are met:

- Sets the cancel bit (UCB$V_CANCEL) in the UCB's status word (UCB$W_STS).

Remember:

- Only I/O requests queued for the calling process are canceled by EXE$CANCEL.
- Canceling queued I/O requests occurs before the driver cancel I/O routine is entered.
- I/O requests queued for file-structured devices are not canceled.
- Current I/O requests for DMA devices cannot be canceled. Therefore, drivers for DMA devices often do not have a driver cancel I/O routine.

INITIALIZATION ROUTINES

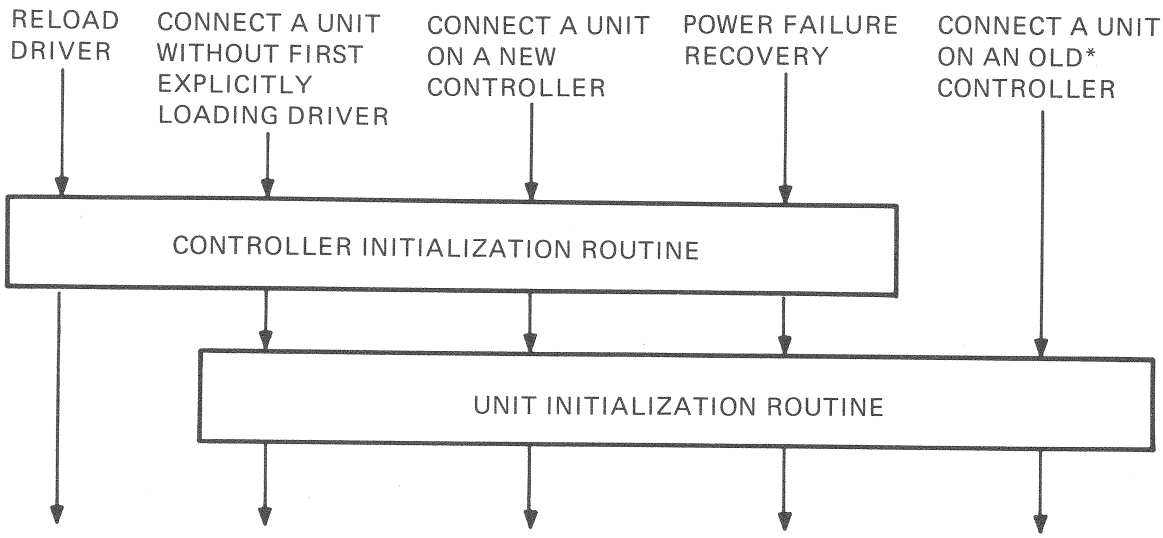
Initialization routines ready controllers and device units for operation. They may perform some of the following operations:

- Enable Interrupts.
- Clear error status bits in device registers.
- Initiate a device operation (for example, clear a drive, or acknowledge a pack).
- Store values in data structure fields that cannot be addressed with the DPT_STORE macro (which can only address the first 256 bytes of a data structure and which is discussed in the Driver Tables module).
- Permanently allocate resources such as mapping registers and/or data paths.
- Set the on-line bit (UCB$V_ONLINE in UCB$W_STS).
- Fill in the IDB$L_OWNER field for single unit devices.

Initialization routines are called at IPL=31, and must not lower IPL. The routines should check to see if they are being called after a powerfailure (by testing the UCB$V_POWER bit in UCB$W_STS) if they permanently allocate resources (to avoid reallocating resources, and never giving resources back to the system).

Normally, the controller initialization routine modifies the IDB and CRB data structures, if necessary, and the unit initialization routine modifies fields in the UCB, if necessary.

After a powerfailure, both unit and controller initialization routines are called. When a driver is first loaded into the system, the controller initialization routine is called. For each unit 'connected' to the driver, the unit initialization routine is called. If a driver is 'reloaded,' only the controller initialization routine is called (see Figure 3-17).



NOTE:
IF YOU ALLOCATE PERMANENT RESOURCES IN THESE ROUTINES, BE CAREFUL ON POWER FAILURE RECOVERY NOT TO DOUBLY ALLOCATE THESE RESOURCES. CHECK THE POWER FAILURE BIT UCB$V_POWER IN UCB$W_STS.

* A CONTROLLER WHICH ALREADY HAS A UNIT CONNECTED TO IT

TK-4829

Figure 3-17 Driver Controller and Unit Initialization Routines

digital

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