EDUCATIONAL
SERVICES

VAX/VMS
Training

VAX/VMS
Device Driver
Overview

digital

EY-2278E-MA-0001

OVERVIEW

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

TABLES

INTRODUCTION

A device driver is a set of routines and tables that the operating system uses to process an I/O request for a particular device type. This module begins by reviewing some key VMS operating system concepts. The module then introduces the primary components of a device driver, and describes some of the basic concepts associated with device drivers. Key terms are highlighted, and definitions for important concepts are provided.

The topics mentioned in this module are expanded upon in later sections of the course. The major intent of this module is to present the general concepts around which this course is built.

OBJECTIVES

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

1. Define what is meant by the following terms:

   - device driver
   - direct memory access
   - programmed I/O
   - driver fork process
   - fork dispatcher
   - mapping register
   - data path
   - interrupt priority level

2. Distinguish the following closely related topics:

   - buffered/direct/programmed I/O
   - buffered/direct data path
   - device dependent/independent parameters
   - driver/driver fork process
   - wait/fork queue

3. Diagram the basic organization of a device driver.

4. List the primary functions of a device driver.

RESOURCES

1. Guide to Writing a Device Driver for VAX/VMS
2. VAX-11 Software Handbook

TOPICS

I. VMS REVIEW

- Processes, System Routines
- IPL, Access Mode
- Linked Exec vs I/O Components

II. DEVICE DRIVERS

- Functions
- Components
- Organization

III. I/O BASICS

- Architecture
- Data transfer methods
- Types of I/O
- $QIO

IV. FLOW AND MECHANISM

- I/O sequence
- Fork Processes, Queues and Dispatching
- Driver Resources

VMS REVIEW

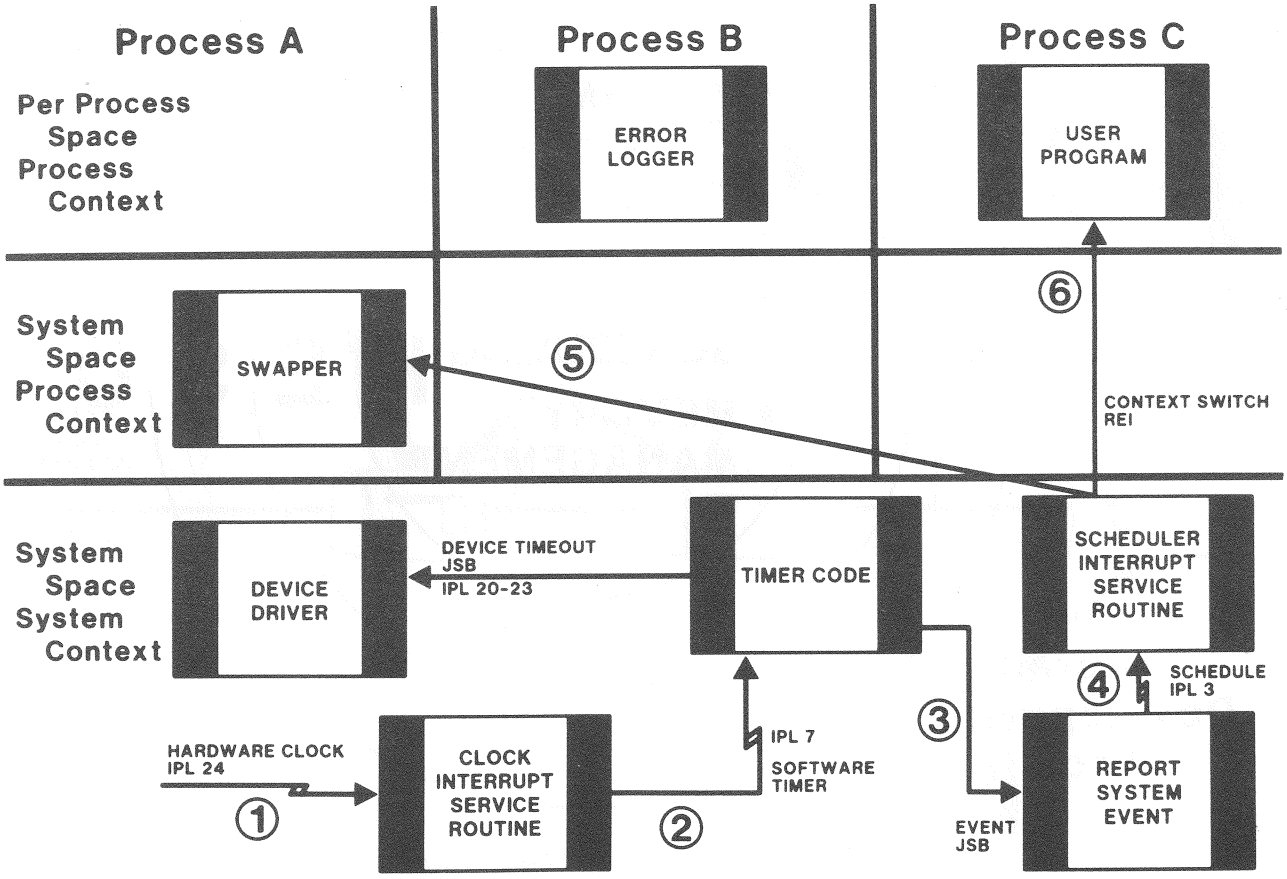

Figure 1-1 System Components

Processes

P0, P1, S0 Space
Process Context

System Routines

S0 Space
System Context

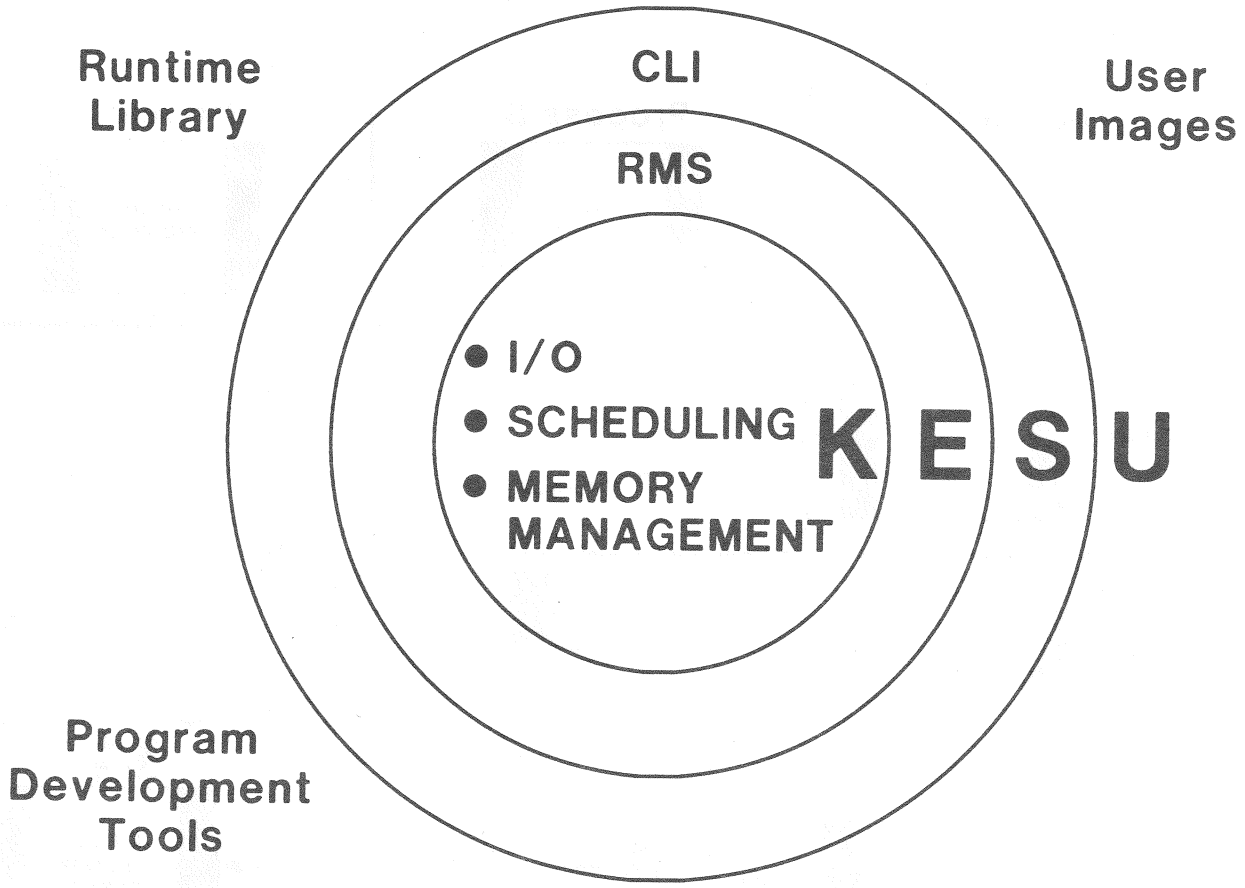

Figure 1-2 Access Modes

Access Modes

User, Supervisor, Executive, Kernel

IPL

Interrupt Priority Level

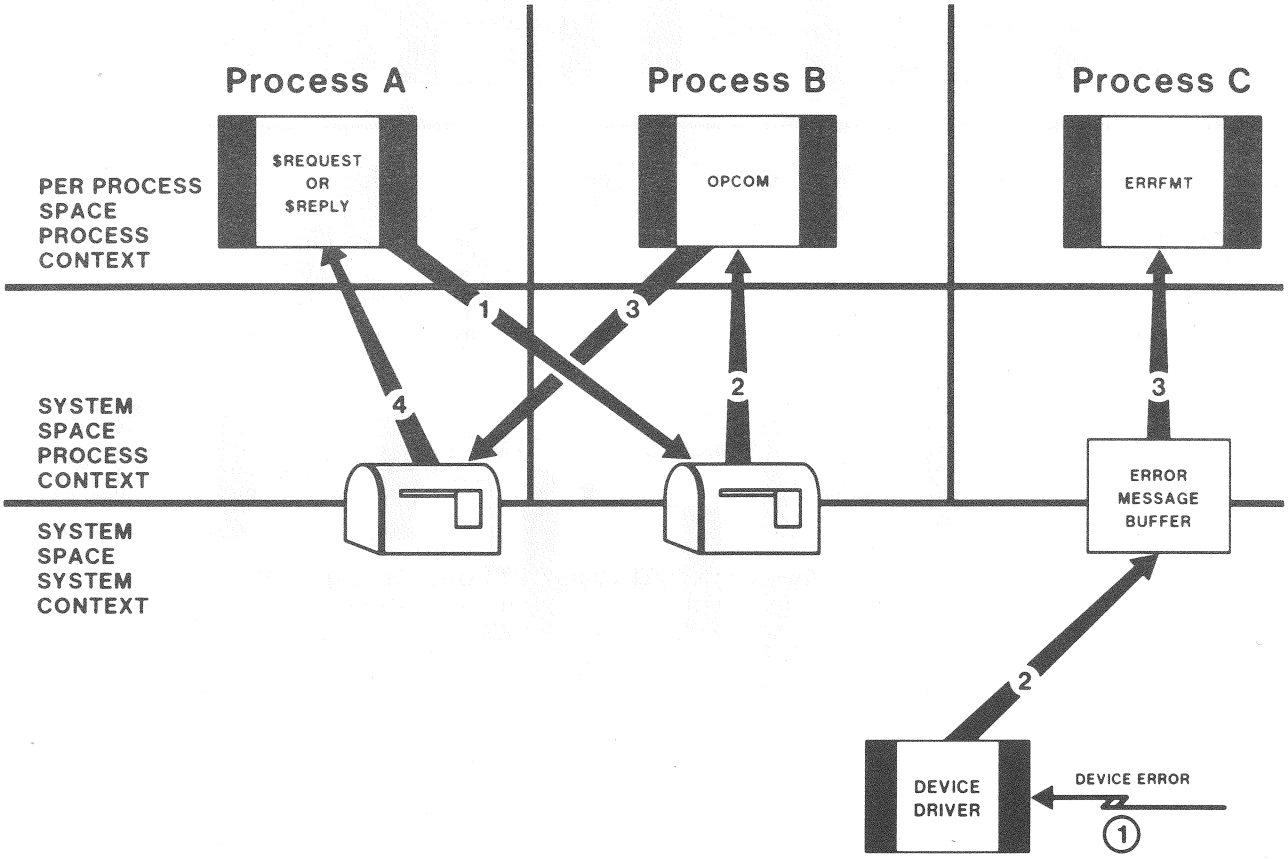

Figure 1-3 Communication

- VMS
  - Linked Executive (SYS.EXE)
  - System Processes (i.e., OPCOM, ERRFMT)
- Communication
  - Mailboxes
  - Global Sections

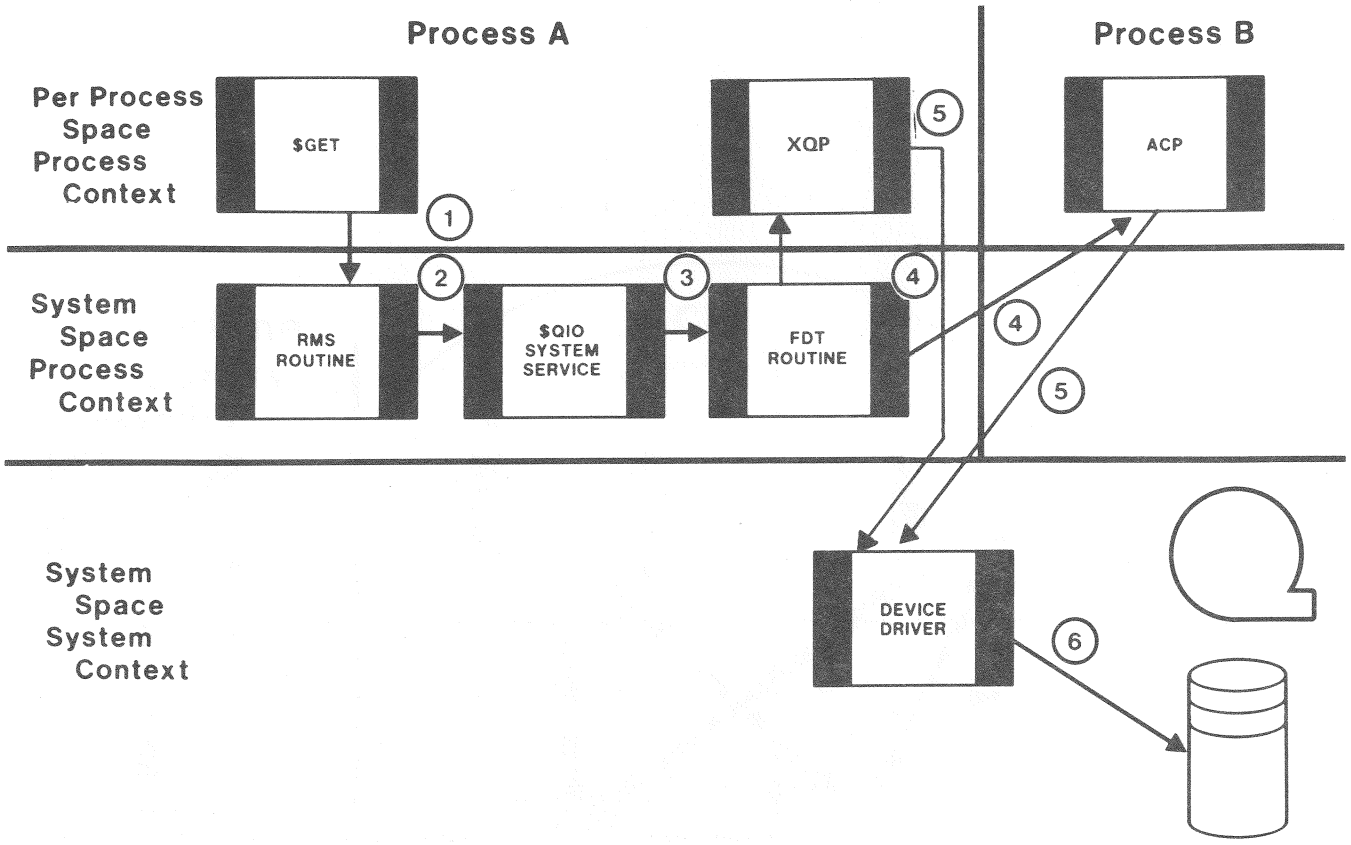

Figure 1-4 Input/Output Flow Using RMS

I/O

- Initiated by User Process
- Preprocessed by
  - RMS
  - $QIO
  - FDT (Driver related routines)
- ACP
  - Disk Structure (FILES-11 ODS-1)
  - Tape Structure
- XQP
  - Disk Structure (FILES-11 ODS-2)

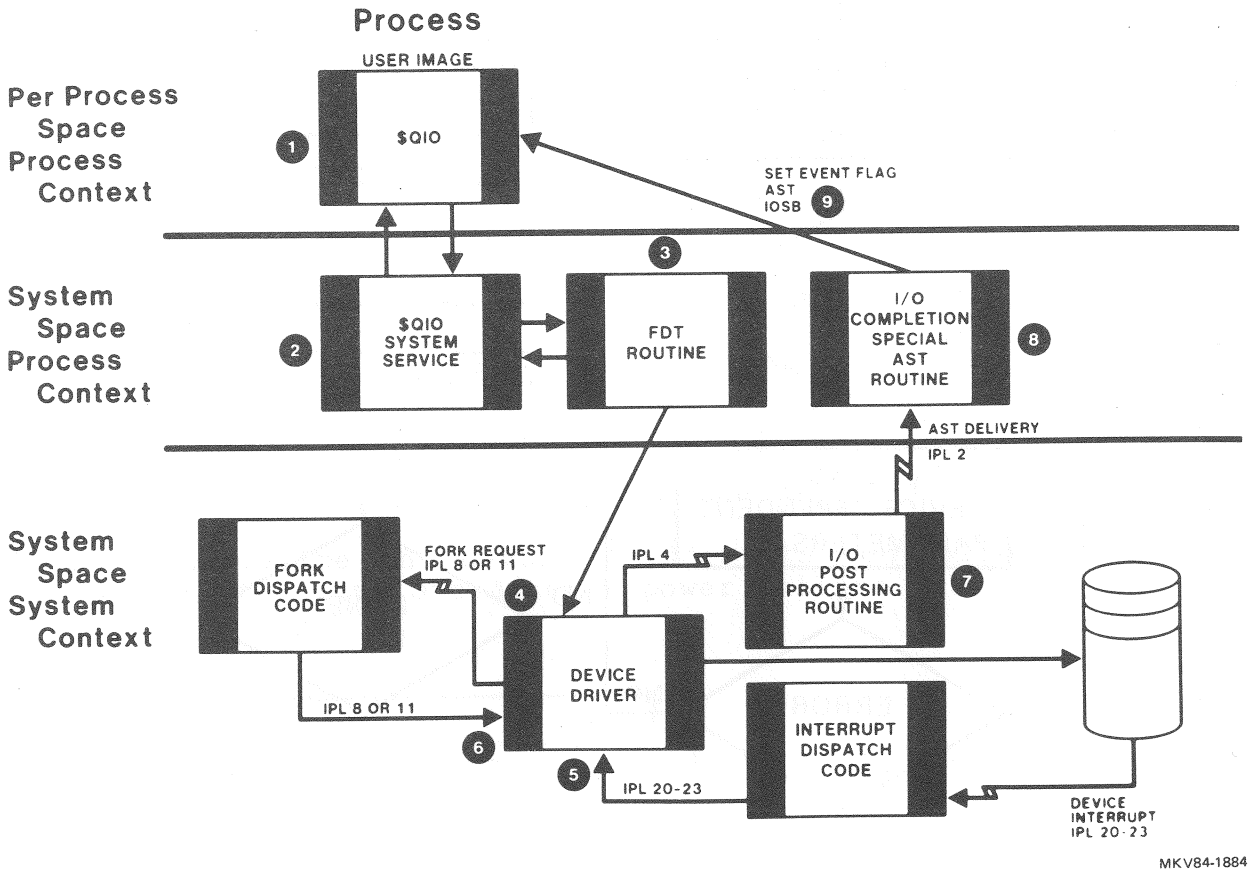

Figure 1-5 Input/Output Flow (Full)

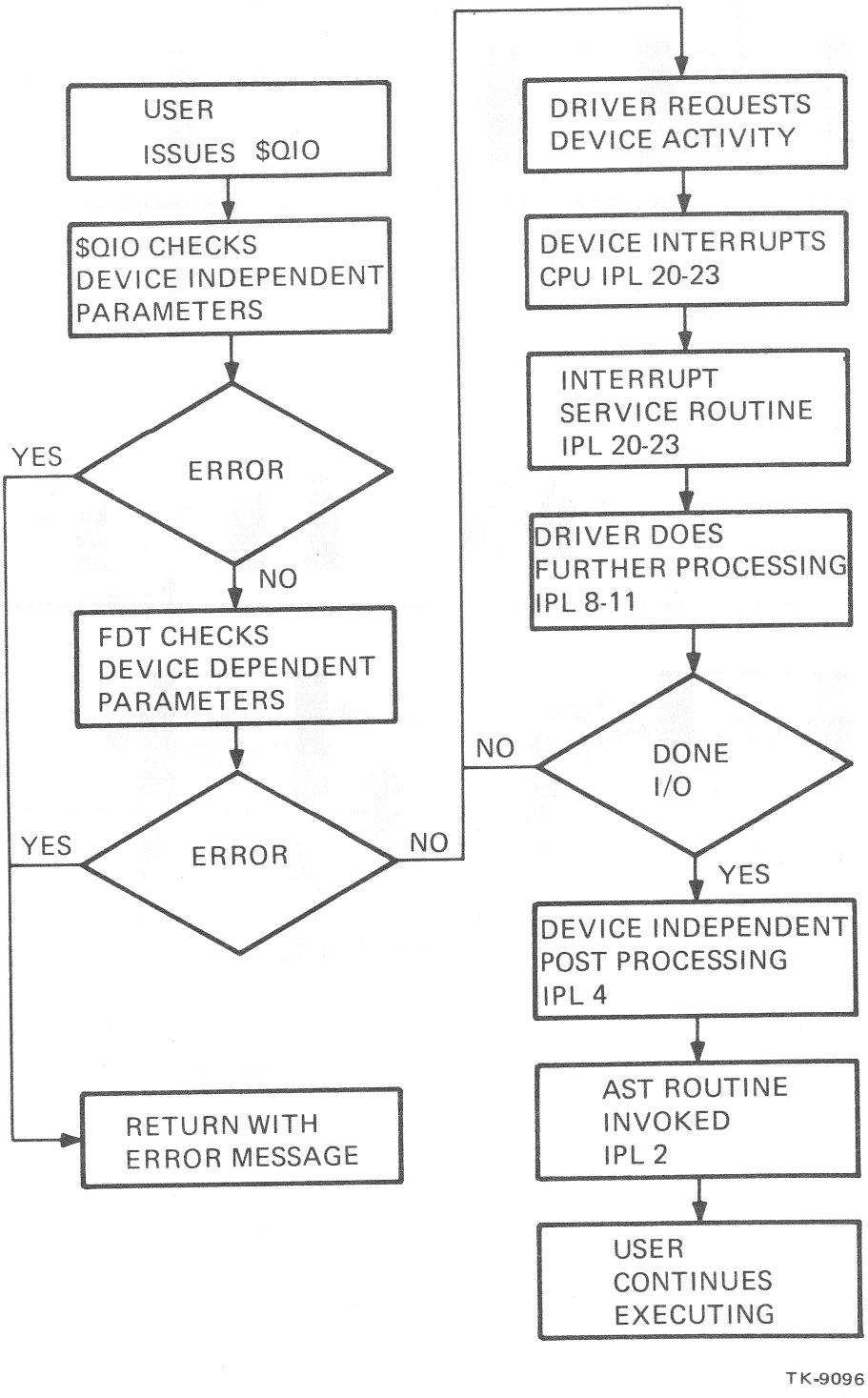

Figure 1-6 I/O Flow Chart

THE DEVICE DRIVER

A **device driver** is a set of tables and routines that control I/O operations on a peripheral device (see Figures 1-7 and 1-8).

Driver Tables

- Define the peripheral device for the rest of the operating system.
- Define the driver for the operating system procedure that maps and loads the driver routines and data structures into system virtual memory (SYSGEN).

Driver Routines

- Initiate device input or output.
- Respond to hardware interrupts generated by the device.
- Report device errors.
- Return data and status from the device to user software.
- Initialize the device (and/or its controller) at system startup time and after a power failure.

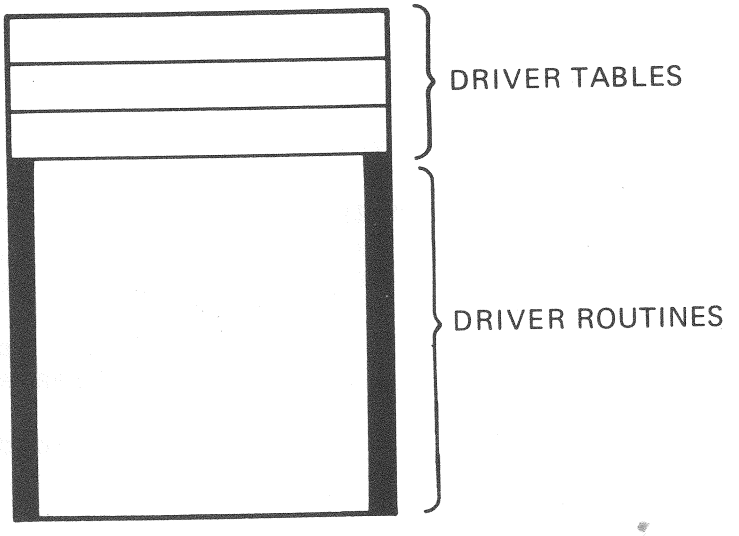

Figure 1-7 Structure of a Device Driver

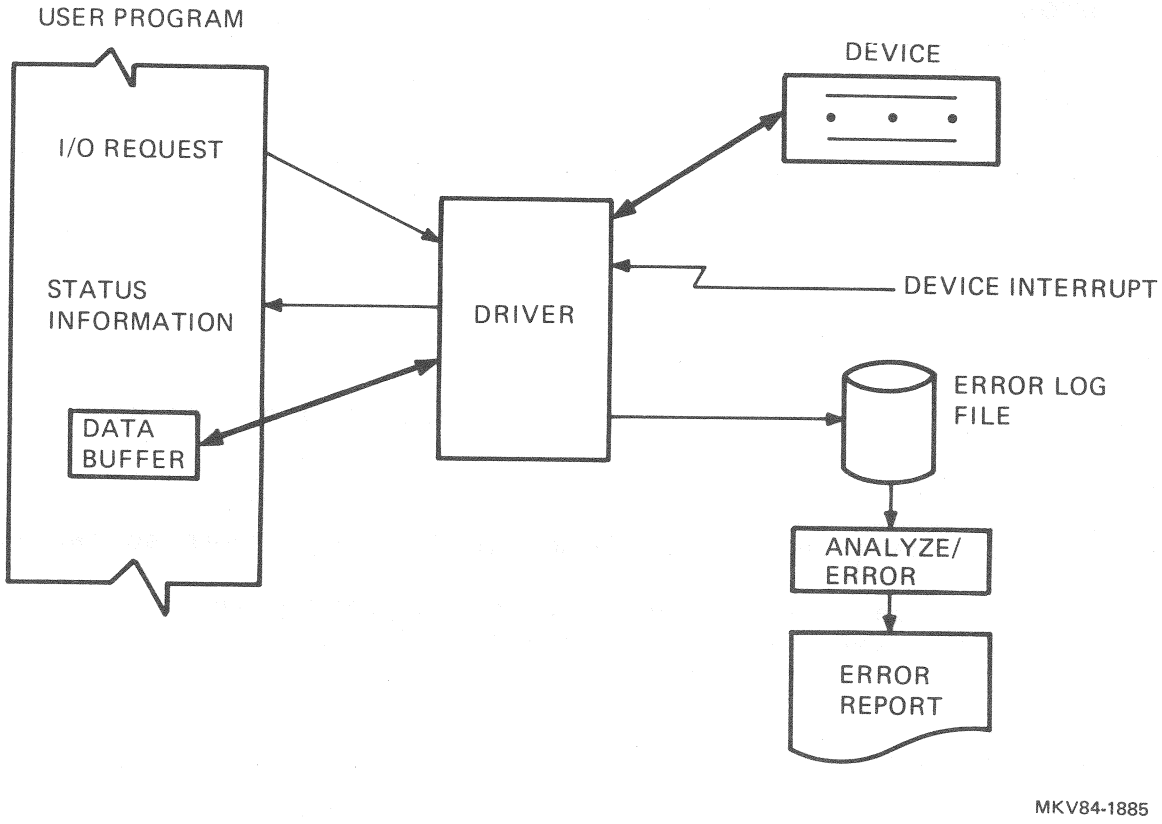

Figure 1-8 User Program, Driver, and Device Relationships

Driver Functions

- Initialize device*

  Controller
  Unit

- Validate user I/O requests
- Start I/O on device
- Handle expected device interrupts
- Handle unexpected device interrupts
- Respond to device errors
- Handle device timeouts
- Log device errors*
- Cancel I/O operations on device*
- Complete I/O requests

* Drivers need not perform these functions.

Driver Tables

Every driver contains the following three tables (see Figure 1-9):

- **Driver Prologue Table (DPT)** - Describes the driver (name, size, etc.) and device type to SYSGEN, the operating system driver loading procedure.
- **Driver Dispatch Table (DDT)** - Lists the entry point addresses of the various driver routines.
- **Function Decision Table (FDT)** - Lists all valid $QIO function codes for the device. Also lists the entry point addresses for I/O preprocessing routines for each $QIO function code (referred to as **FDT routines**).

Driver Organization

Figure 1-9 illustrates the general organization of a device driver. By consulting the tables in the driver, and by examining various system-wide data structures referred to as the I/O data base, the operating system determines which entry point(s) of the driver to reference to satisfy user $QIO requests (see Figure 1-10). Drivers do not decide when to act or what function to perform. Rather, the operating system is responsible for the decisions.

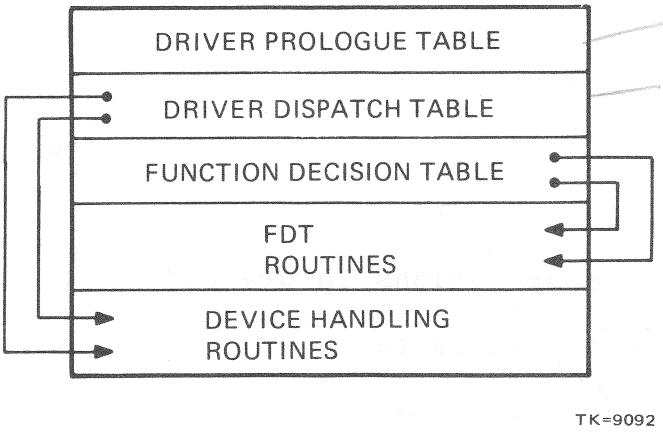

Figure 1-9 General Organization of Device Driver Tables and Routines

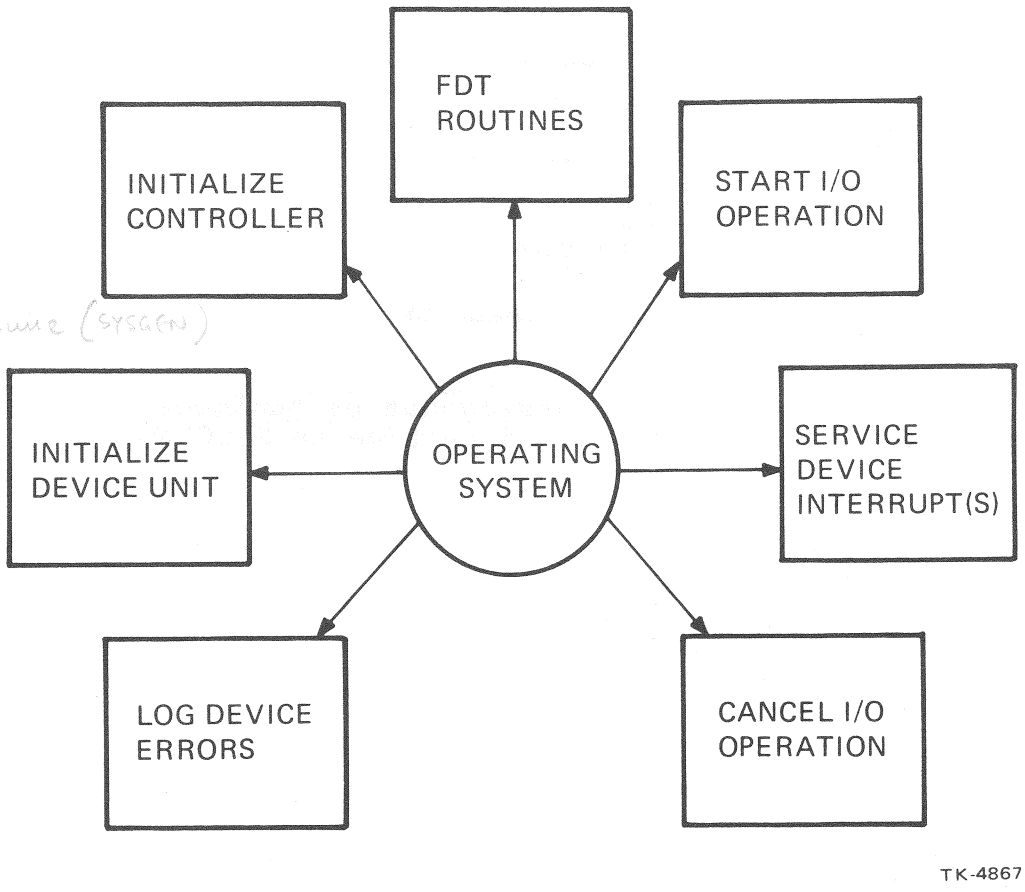

Figure 1-10 The Operating System Decides Which Driver Routines to Call to Satisfy a User QIO Request

Loadable VAX Device Drivers

- Some drivers are linked with the executive.
  - Part of the bootstrapped image
- Other drivers are loaded while the system is running.
  - AUTOCONFIGURE (part of system initialization) loads DEC-supported device drivers.
  - A user can manually load other drivers (often done as part of the site-specific startup command file).
  - SYSGEN utility is used to load drivers. (see Figure 1-11.)
  - Once loaded, drivers cannot be "unloaded." There is, however, a "reload" directive to SYSGEN.

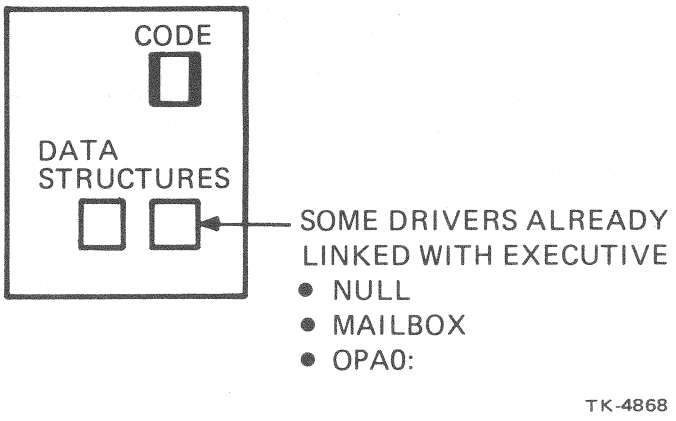

Figure 1-11 Using the SYSGEN Utility to Load Drivers into Nonpaged Pool (Memory)

VAX I/O ARCHITECTURE

- All major system components are connected by a bus called the Backplane Interconnect (see Figure 1-12).
- **UNIBUS** - a medium-speed bus for terminals, smaller disks, and user peripherals.
- **MASSBUS** - a high-speed bus for large disks and magtapes.

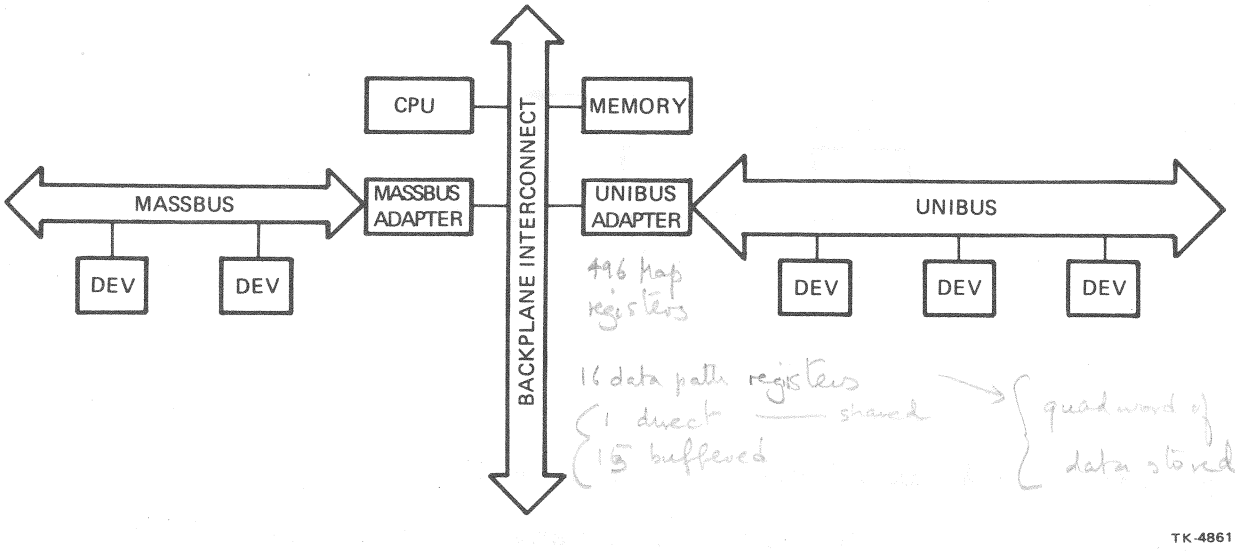

Figure 1-12 The Backplane Interconnect Connects All Major System Components

DATA TRANSFER METHODS

- A hardware distinction

Programmed I/O (see Figure 1-13)

- CPU must get/put each byte or word.
- Device generates an interrupt after each byte or word.

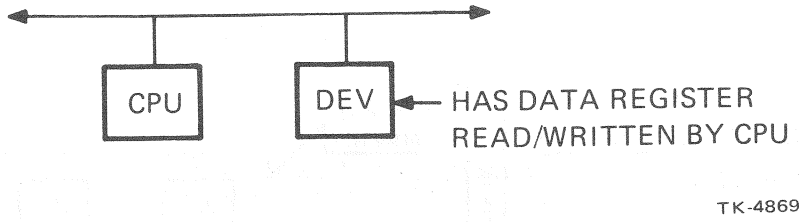

Figure 1-13 Programmed I/O Device

Direct Memory Access (DMA) (see Figure 1-14)

- CPU tells the device how many bytes or words to transfer.
- CPU tells the device where the user's buffer is in memory and sets GO bit.
- Device transfers a chunk of data (not necessarily 512 bytes).
- Device communicates directly with memory.
- Device generates interrupt at the end of the block.

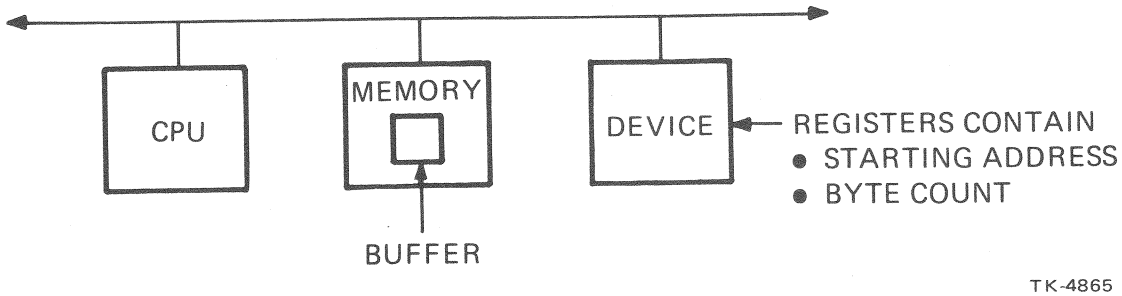

Figure 1-14 Direct Memory Access Device

PDP-11 UNIBUS ADDRESS SPACE AND REGISTER CONCEPTS

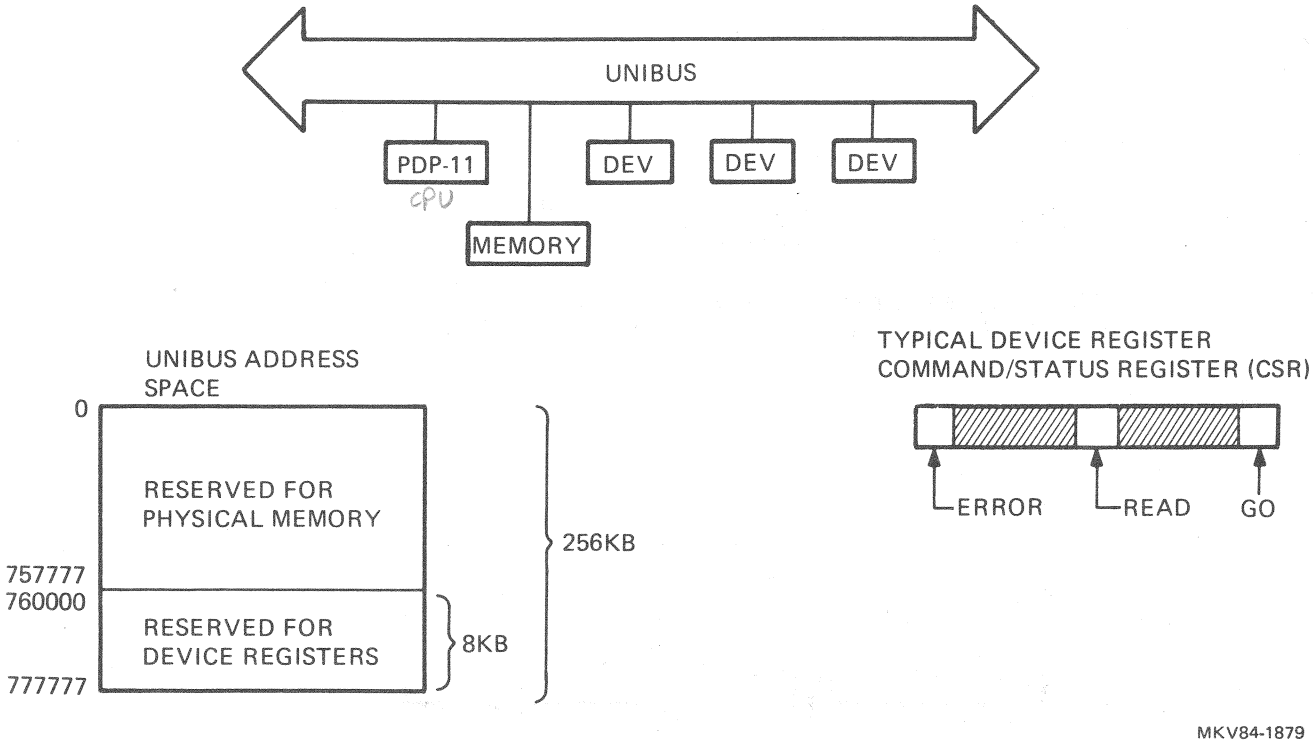

Figure 1-15 UNIBUS Address Space

- Address space split into two parts.
- Device Registers have physical addresses.
- Control of device and information is passed through device registers.

TYPES OF I/O

- A software distinction

Direct I/O (see Figure 1-16)

- Data is transferred directly between user buffer and device.
- Buffer is not normally accessible to driver's start I/O and interrupt routines.
- Buffer pages are locked (not swappable or pageable) until the I/O completes.
- Useful mainly for DMA devices.

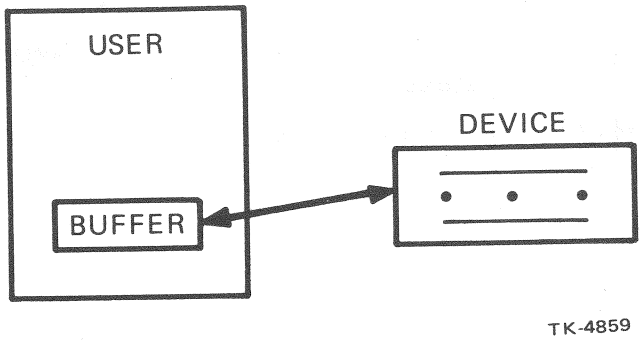

Figure 1-16 Direct I/O

Buffered I/O (see Figure 1-17)

- Data is transferred through an intermediate buffer in the system address space (SØ).
- User process is completely swappable and pageable.
- Useful for both DMA and programmed devices.

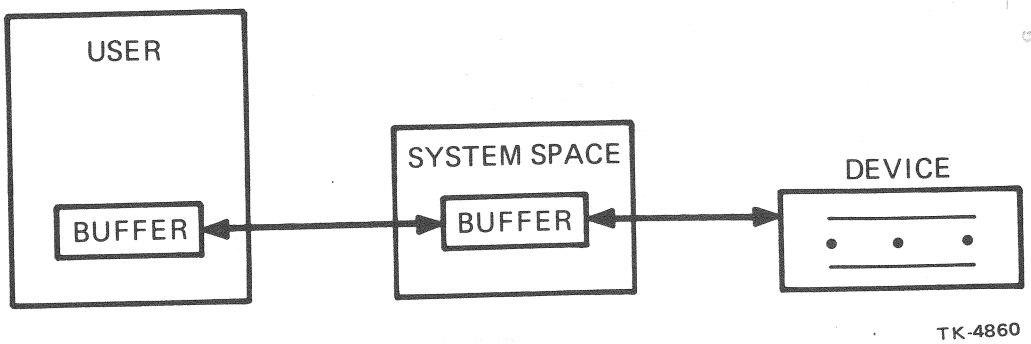

Figure 1-17 Buffered I/O

$ASSIGN SYSTEM SERVICE

- Establishes a software link (channel) between a user process and a device (see Figure 1-18).
- $ASSIGN devnam, chan, ---
  - User specifies device name.
  - System returns a channel number.

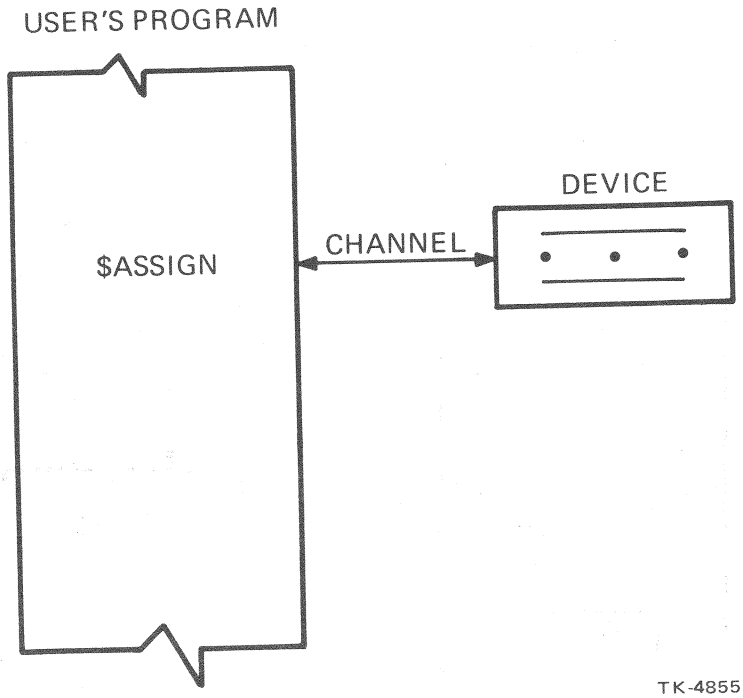

Figure 1-18 Formation of a Channel Using the $ASSIGN System Service

$QIO SYSTEM SERVICE

- Queues I/O requests to driver (see Figure 1-19).
- $QIO ..., chan, func, iosb, ..., P1, P2, ..., P6

chan, func, iosb, ...	P1, P2, ..., P6
• Device/function independent	• Device/function dependent
• Checked and handled by the system	• Checked and handled by the driver FDT routines

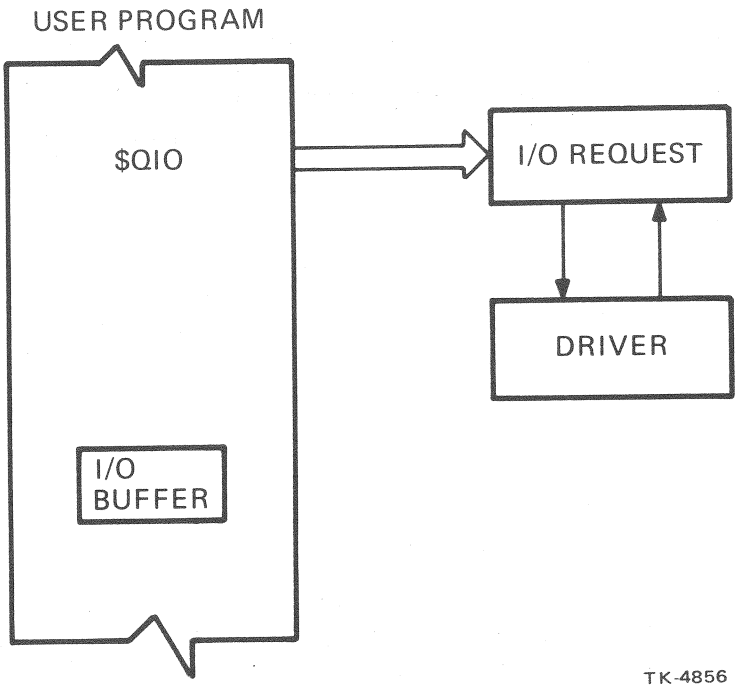

Figure 1-19 I/O Requests Queued to a Driver

All user requests for device activity are made through the $QIO system service (either explicitly, or implicitly via RMS calls). The format of the $QIO system service is as follows:

$QIO[W] [efn], chan, func, [iosb], [astadr], [astprm], [P1], [P2], [P3], [P4], [P5], [P6]

The first six parameters are **device-independent parameters,** since they have the same meaning for all devices. The P1-P6 parameters are **device-dependent parameters,** since they are interpreted differently for various devices.

The writer of a device driver decides which (if any) of the P1-P6 parameters are to be used (for specific function codes), and what information is to be placed into P1-P6. P1 is typically considered an address, while P2-P6 are interpreted as values. (The writer of a device driver also decides which function codes are valid for the device, and what those function codes cause the device to do.)

It is the responsibility of the FDT routines to interpret and record the device-dependent parameters (P1-P6) for later processing by the driver routines.

DRIVER FORK PROCESSES

I/O operations are handled by driver fork processes. A **driver fork process** is created dynamically, and has minimal context (See Figure 1-20). The context of a driver fork process consists of:

- registers R3 and R4.
- a data structure called a Unit Control Block, which describes the device for which the $QIO is issued (pointed to by R5).
- the Program Counter (PC) for the next instruction to execute in the driver.
- system virtual address space.

Like user processes, driver fork processes can be suspended and interrupted. Driver fork processes are placed in a **wait queue** when they request an unavailable resource. When a driver fork process is placed into a wait queue, its context is "folded" into the Unit Control Block (i.e., registers R3 and R4 are copied into fixed fields of the Unit Control Block, along with the PC of the instruction to execute when the resource becomes available). Once the resource becomes available, the driver fork process context is restored, and execution continues (R3 and R4 are retrieved from the Unit Control Block, as is the PC for the next instruction, and the address of the Unit Control Block is placed in R5).

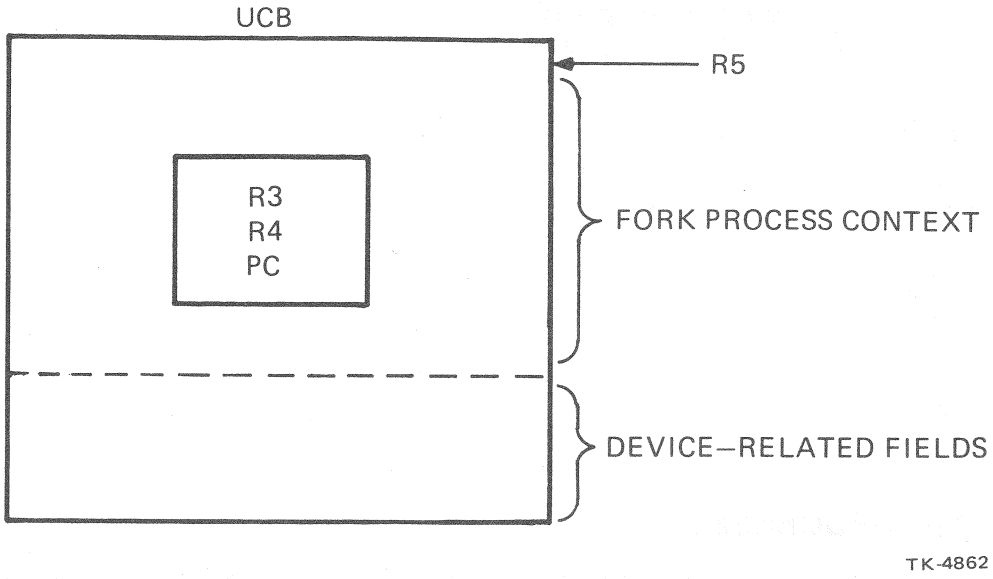

Figure 1-20 Fork Process Context

Fork Process

- A "process" which runs in SØ space.
- Must remain in SØ since it does not know which user process is current (see Figure 1-21).
- Runs at IPL 6,8,9,10, or 11 until:
  - Must wait for a resource.
  - Voluntarily relinquishes control (enters wait state).
- One fork process for each busy unit.
- Exists only as long as I/O request is being serviced.
- Created when I/O request is queued to device that is not busy.

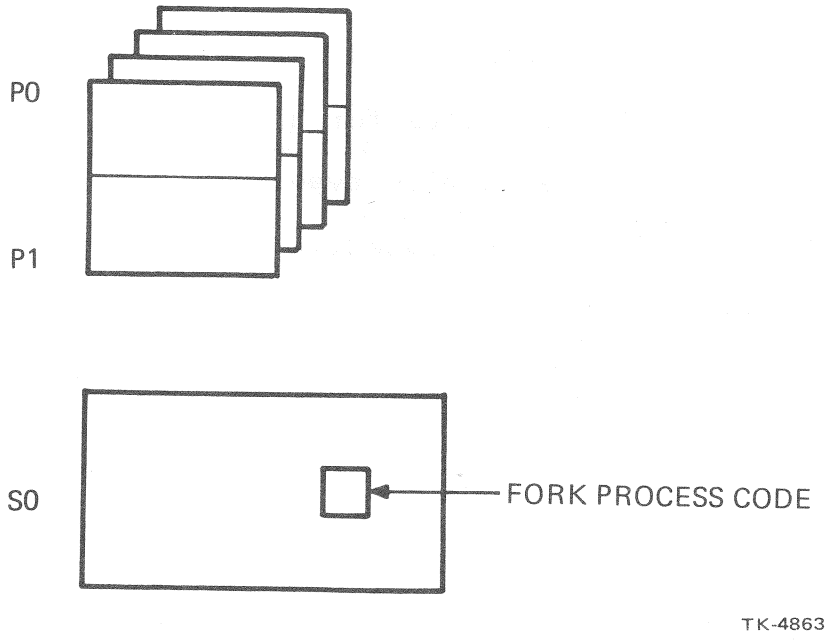

Figure 1-21 Fork Processes Execute in System Address Space

Driver

- Is part of the executive.
- Contains reentrant code and data structures.
- Handles all I/O requests for a device type.
- Has one driver per device type.
- Stays in system once loaded.

Driver Fork Process

- Is an execution agent with minimal context that is created dynamically.
- Executes driver code on behalf of a device unit.
- One driver fork process per device unit that is busy.
- Is created when an I/O request is queued for a nonbusy device unit.
- Disappears after the I/O operation is complete.

NOTE: A driver and driver fork process are separate entities. The terms are NOT synonymous.

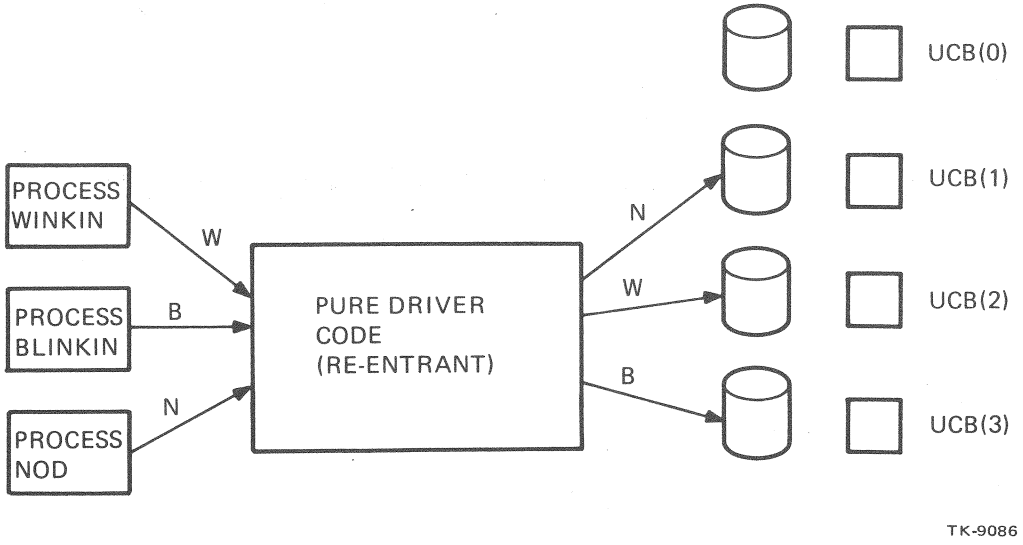

Figure 1-22 Driver and Driver Context

Interrupts

- Interrupts cause the current PC, PSL, and other information to be pushed on the interrupt stack (see Figure 1-14).

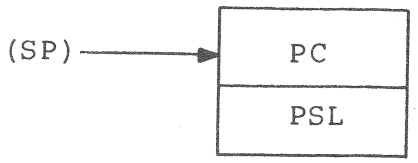

Figure 1-23 Pushing PC and PSL onto the Interrupt Stack

- Microcode vectors through System Control Block (SCB) to the appropriate interrupt service routine (see Figure 1-24).
  - SCB has entries for all possible exceptions and interrupts.

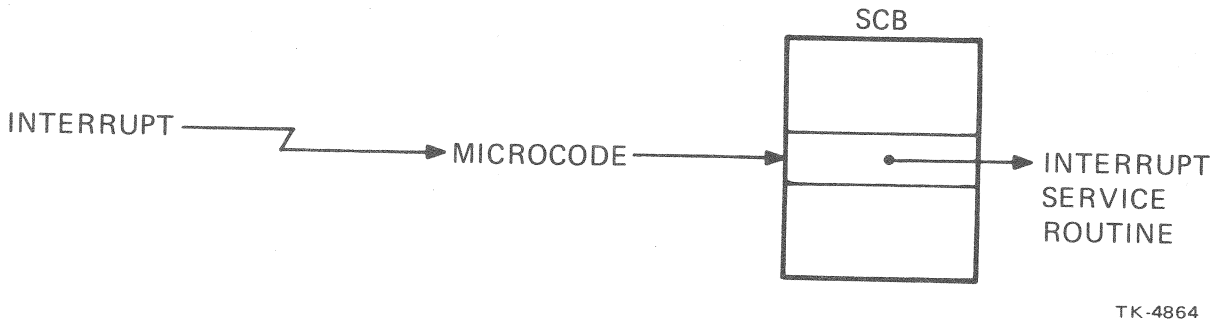

Figure 1-24 All Interrupt Service Routines Are Found in Fixed Locations in the SCB

- Interrupts occur at 31 interrupt priority levels (IPLs).
  - CPU can set IPL to mask interrupts.
  - CPU running at IPL i masks interrupts at and below i.
  - Interrupt at IPL i raises CPU to IPL i.

Synchronization

To synchronize the execution of drivers within the system, and to synchronize access to shared system resources, VMS employs a scheme involving IPLs.

- IPL is a hardware-maintained quantity.
- There are 32 levels of IPL.
- Higher-numbered (16-31) IPLs are reserved for hardware interrupts.
- Lower-numbered (1-15) IPLs are reserved for software interrupts.
- A high IPL takes precedence over a lower IPL.
- IPL Ø is for normal process execution, and is not considered an interrupt level.
- User processes have a software priority (also a number between Ø and 31 associated with them); this software priority has no relation to IPL.
- Driver fork processes execute at IPL 6 and IPLs 8-11 (referred to as **fork IPLs**).
- Devices interrupt at IPLs 2Ø-23 (referred to as **device IPLs**).

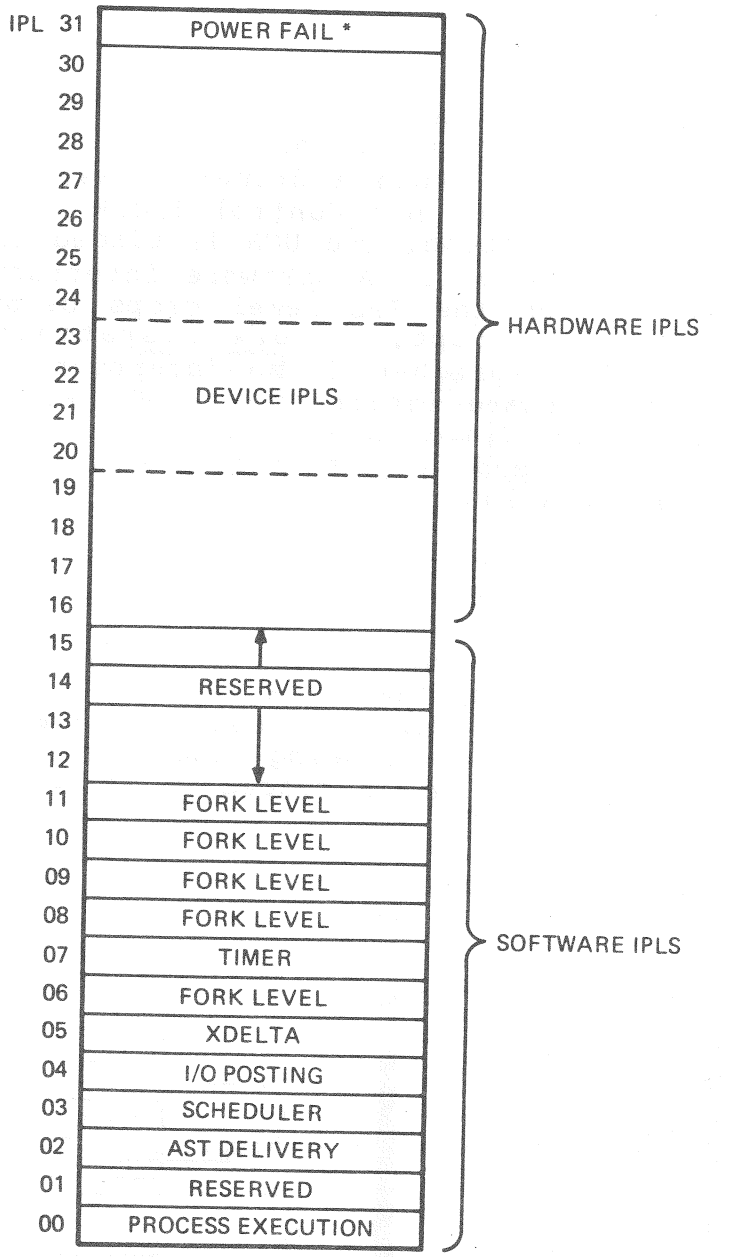

Figure 1-25 Interrupt Priority Levels

Forking

Drivers try to run at the lowest possible IPL at all times; therefore, they occasionally ask to have their IPL lowered (called **forking**; see Figure 1-26). When a driver forks, the contents of R3 and R4 are placed in the Unit Control Block (UCB), along with the PC of the next instruction. The UCB is placed on a **fork queue** for the appropriate IPL level. A software interrupt is requested for the IPL level. When the IPL level drops below that of the requested software interrupt, a **fork dispatching** routine is activated. (The fork dispatcher is an interrupt service routine which responds to a software interrupt at fork IPL levels 6, and 8 through 11.) The fork dispatching routine restores the driver fork process context, and allows the driver fork process to continue execution (at a lowered IPL).

Fork queues and wait queues are similar in that the context of the driver fork process is stored in the UCB. However, the two queues differ in that driver fork processes are placed into wait queues because they need an **unavailable resource** (when they resume execution, they remain at the same IPL), while driver fork processes are placed into fork queues (to **lower IPL**), and no resource allocation is involved. Being placed in a fork queue is a voluntary action, whereas being placed in a wait queue is an involuntary action.

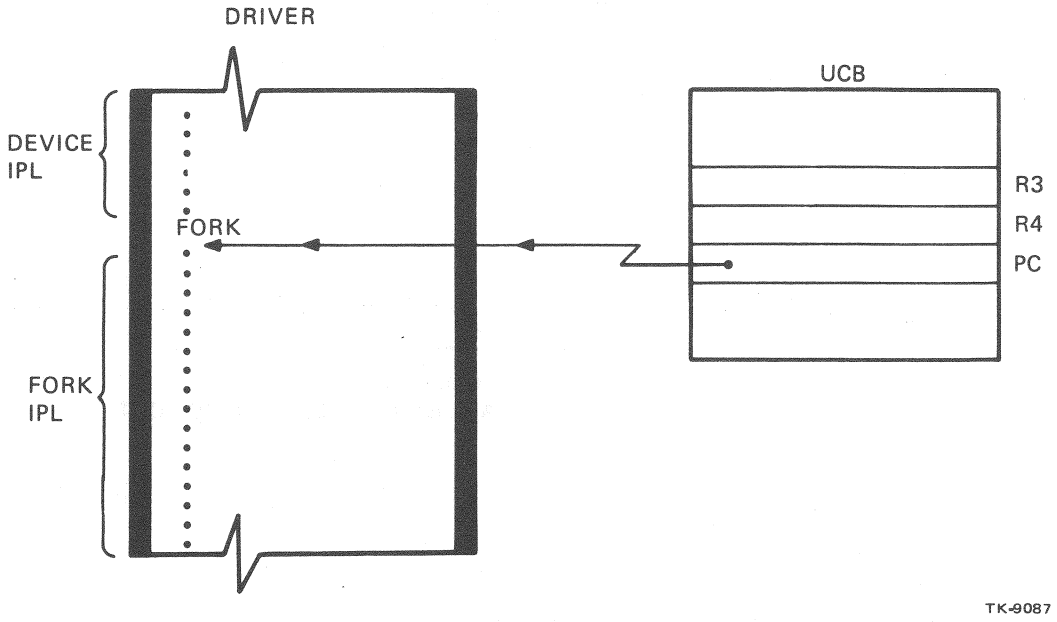

Figure 1-26 Driver Fork Context

Fork Dispatching

- Fork process wants to lower IPL.
- Context is stored in UCB.
- UCB is queued to listhead for new IPL (IPL 9 in Figure 1-27).

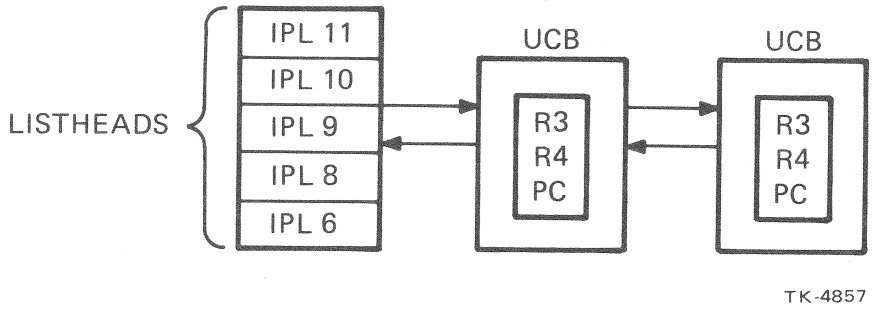

Figure 1-27 Fork Queue Listheads

- Software interrupt requested at new IPL.
- Serializes all fork processes running at same IPL.
- Fork dispatcher gains control (as interrupt service routine for some IPL, e.g., IPL 9 in Figure 1-28).

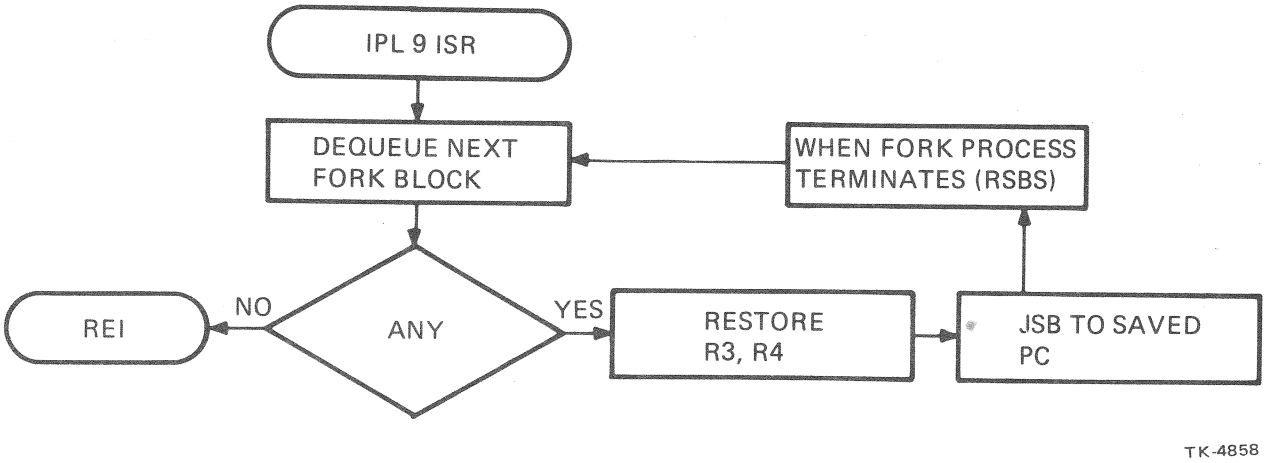

Figure 1-28 General Logic Flow in Fork Dispatcher

Fork Queues and Wait Queues

Fork Queue

- One fork queue per fork IPL.
- Queue of driver fork processes waiting to be resumed by the fork dispatcher.
- First in, first out queue.
- Placed in the queue in order to lower IPL.

Fork Dispatcher

- Gains control of software interrupts at levels 6, and 8-11.
- Is really an interrupt service routine.
- Resumes driver fork processes in fork queue corresponding to interrupting IPL.

Wait Queue

- Queue of driver fork processes waiting for a particular resource.
- Driver fork processes in queue are resumed when resource becomes available (fork dispatcher is not involved).
- First in, first out queue.

A driver fork process may be compared to a normal process in several ways. Table 1-1 lists several analogies that can be made between process context and driver fork process context.

Table 1-1 Process Context and Driver Fork Process Context

Process Context	Driver Fork Process Context
Process Control Block (PCB)	Unit Control Block (UCB)
P0, P1, S0, Space	S0 Space
IPL $\leq$ 2	IPL > 2
Perprocess Stack	Interrupt Stack
Synchronization via: Mutexes Raising IPL	Synchronization via: Wait Queues Raising IPL Forking
Can be Placed in: Scheduler Wait State	Can be Placed in: Resource Wait Queue Fork Queue

DRIVER RESOURCE CONTENTION

Drivers compete for the following shared resources:

- Central processor
- Mapping registers
- Buffered Data Paths
- Controller Data Channel (if there is more than one device unit attached to a controller)

Drives for real devices execute at fork IPL 8 because there is no explicit synchronization for fork processes that compete for adapter resources.

Mapping Registers

Mapping Registers are used by both the UNIBUS adapter and the MASSBUS adapter. Each adapter has its own set of mapping registers. They are used to translate UNIBUS/MASSBUS addresses, which the device understands, into addresses the system understands. (The translation scheme is discussed in detail in the I/O Architecture module.) There are a limited number of mapping registers per adapter.

Data Paths

The UNIBUS adapter sends data through one of two kinds of data paths for UNIBUS devices performing Direct Memory Access (DMA) transfers. (The MASSBUS has no data path concept.) The direct data path can be used by any device at any time. A buffered data path must be allocated to a device before it can be used. Do not confuse direct and buffered data paths with direct and buffered I/O. Note also that data paths only have meaning for DMA operations on the UNIBUS.

- Direct Data Path - only one per UNIBUS adapter. Allows 8 or 16 bits (1 UNIBUS transfer) of data to be transferred to system memory at a time.

- Buffered Data Paths - 15 per UNIBUS adapter. Allows up to 32 or 64 bits (4 UNIBUS transfers) of data to be transferred to system memory at a time.

Data paths are described in detail in the I/O Architecture module, where guidelines are also given concerning the decision to use the buffered or direct data paths. In general, buffered data paths are more efficient than the direct data path, and should be used whenever possible. Table 1-2 illustrates the typical use of data paths for devices performing programmed I/O and DMA.

Controller Data Channel

Since only one device unit can transfer data on a controller at a given time, driver fork processes must contend for ownership of the controller data channel.

Table 1-2 I/O Methods Used for Programmed I/O and DMA Data Transfers

Data Transfer Method	I/O Used	Data Path Used (UNIBUS Only)
Programmed I/O	Buffered I/O	None (device registers)
Direct Memory Access	Direct I/O	Buffered Data Path

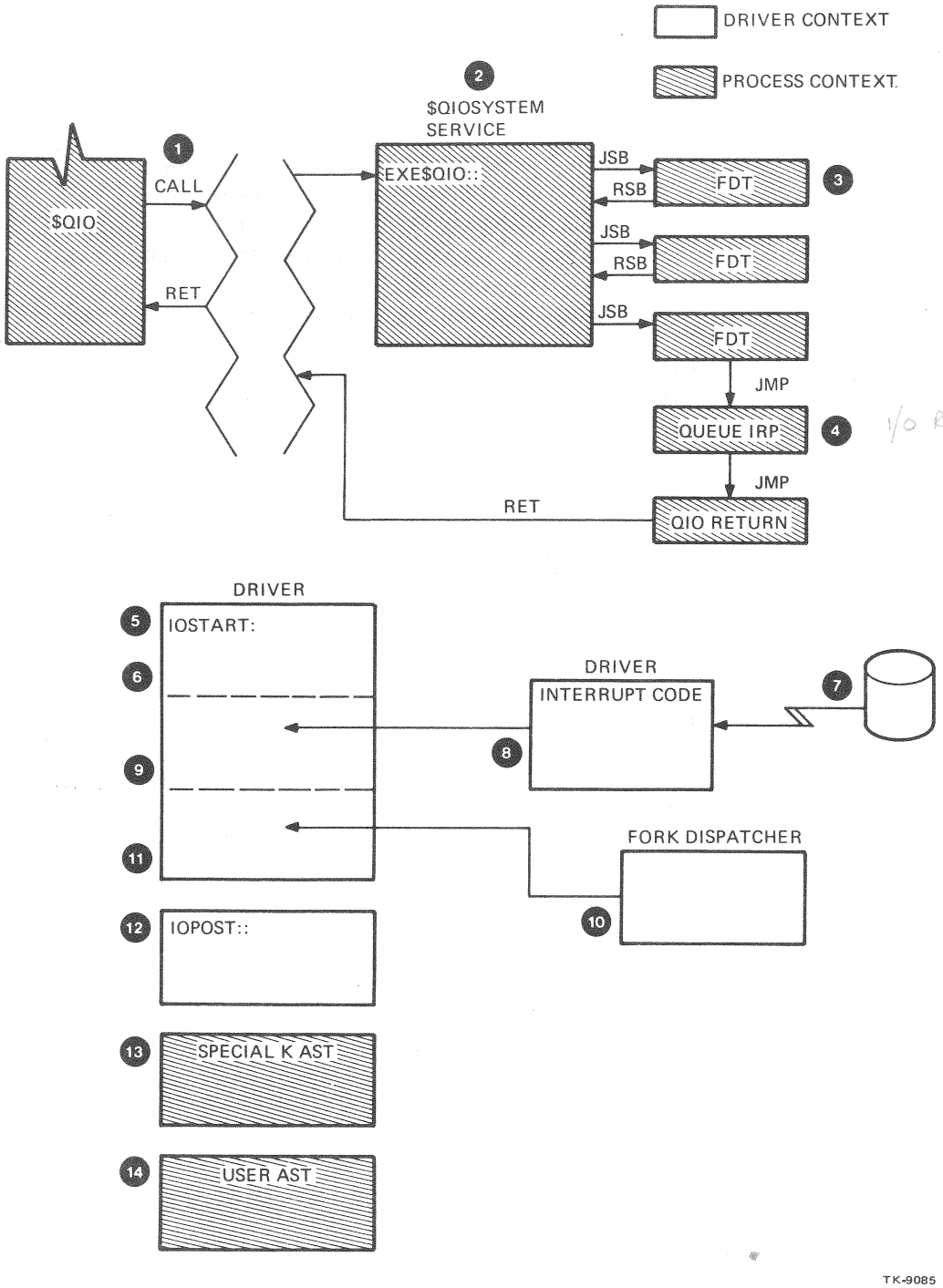

Figure 1-29 I/O Sequence Overview

I/O Sequence

PC	1	User issues $QIO
PC	2	$QIO system service
PC	3	$QIO JSBs to FDT (preprocessing) routines
PC	4	IRP queued to driver (data structures)
SC	5	Driver starts processing I/O request
SC	6	Once I/O is started, driver waits for interrupt
SC	7	Device generates interrupt
SC	8	Interrupt dispatch code returns control to driver (at DIPL)
SC	9	Driver does necessary work, then forks to fork IPL
SC	10	Fork IPL taken, fork dispatcher returns control to driver
SC	11	Driver cleans up, queues IRP to I/O post-processing
SC	12	I/O post cleans up, sets up Sp K Ast, Rei
PC	13	Sp K Ast taken, event flag set, IOSB filled in, user Ast declared
PC	14	User Ast taken

PC = Process Context
SC = System Context

digital

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