VMS

digital

DECwindows Device Driver

Order Number: AA–MG28A–TE

VMS DECwindows Device Driver Manual

Order Number: AA–MG28A–TE

December 1988

This manual describes the DECwindows device driver software on workstations that run VMS. It may be used when you write a DECwindows driver for a device connected to a VAX workstation. It describes the DECwindows driver/server architecture, the various drivers, driver components, their routines, macros, and data structures. It also describes the driver/server interface and methods by which a driver and server call and pass information.

Revision/Update Information: This is a new manual; it does not supersede any previous manual.

Software Version: VMS Version 5.1

digital equipment corporation
maynard, massachusetts

December 1988

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 be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA
DDIF
DEC
DECnet
DECUS
DECwindows
DIGITAL
LN03
MASSBUS
PrintServer 40
Q-bus
ReGIS
ULTRIX
UNIBUS
VAX
VAXcluster
VAX RMS
VAXstation
VMS
VT
XUI
digital™

The following are third-party trademarks:

PostScript is a registered trademark of Adobe Systems, Inc.

X Window System, Version 11 and its derivations (X, X11, X Version 11, X Window System) are trademarks of the Massachusetts Institute of Technology.

ZK4735

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

FIGURES

TABLES

Preface

The *VMS DECwindows Device Driver Manual* provides information needed to understand the driver software and system, and to write a DECwindows driver for an input device. The DECwindows software described in this document is designed to run with VMS Version 5.0 or later and is associated with a certain family of workstations specified in this manual. The manual provides DECwindows data structures, routines, and code examples for the programmer.

Intended Audience

This manual is intended for system programmers who are already familiar with VAX processors and the VMS operating system. Although the discussion of the device driver architecture and components applies specifically to DECwindows workstations with keyboard and mouse, the information also applies to other serial input devices. The driver design described is based on Q22-bus CPU/controller type (VCB01/VCB02) hardware.

Structure of This Document

The manual presents the DECwindows architecture and its main components and functions and then describes the driver/server interface, driver entry points, driver services and routines, and information needed to write a driver. The appendixes contain reference material such as the DECwindows data structures and macros.

If you are coding a server, Chapter 2 and Chapter 6 provide required information concerning $QIO calls for programming devices. The manual contains the following chapters:

- Chapter 1 presents the X Window System concept and introduces the components of the DECwindows architecture. It describes the main software components and their functions, hardware relationships, and DECwindows requirements.
- Chapter 2 describes the common driver queue and server interface. It describes the data format of the serial line interface, queue management and communication protocols. It also describes the $QIO common interface and $QIO calls made from the server.
- Chapters 3 and 4 describe the port and class input drivers. Both chapters provide program entry points and information required to write a DECwindows input driver. Chapter 3 describes how to write a port driver and presents the input driver routines that process input data and manage the devices. These driver routines handle interrupts and manage the controller ports. Chapter 4 describes how to write a class driver and presents the routines that process input data and manipulate the input queue.

- Chapter 5 presents common driver program information and routines that provide common DECwindows services. It provides information concerning management of the queue interface to the server, calls for service in other drivers, and $QIO preprocessing. It also describes the FDT routines, organization, and preprocessing services provided.
- Chapter 6 presents output driver program information and the vectored output routines that provide video and cursor image control, operator window control, and device-dependent $QIO services.
- Appendix A describes the data structures that make up the DECwindows I/O subsystem database. Each data structure is shown in a figure and has an accompanying table that defines each field.
- Appendix B presents the macros for all the common and input driver module software. It describes general driver macros, input queue and packet processing macros, and vector generation macros.

Associated Documents

Because the DECwindows software is integrated with VMS, references are often made to the VMS driver software or I/O subsystem that is described in the *VMS Device Support Manual*. If you are writing a DECwindows device driver, refer to both this manual and the *VMS Device Support Manual* for basic driver design. Before reading the *VMS DECwindows Device Driver Manual*, you should have an understanding of the material discussed in the following documents:

- *VMS Device Support Manual*
- I/O-related portions ($QIO) of the *VMS System Services Reference Manual*
- Terminal driver section of *VMS I/O User's Reference Manual: Part I*

You may also find useful some of the material in your workstation's technical manual. Other related information may be found in the following books:

- *VAX/VMS Internals and Data Structures*
- *Guide to Setting Up a VMS System*
- *VMS System Dump Analyzer Utility Manual*
- *VMS DECwindows Guide to Xlib Programming: VAX Binding*
- *VMS DECwindows Xlib Routines Reference Manual*

Conventions

The following conventions are used in this manual:

mouse
: The term *mouse* is used to refer to any pointing device, such as a mouse, a puck, or a stylus.

MB1, MB2, MB3
: MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (The buttons can be redefined by the user.)

PB1, PB2, PB3, PB4
: PB1, PB2, PB3, and PB4 indicate buttons on the puck.

SB1, SB2
: SB1 and SB2 indicate buttons on the stylus.

Ctrl/x
: A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.

PF1 x
: A sequence such as PF1 x indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.

Return
: A key name is shown enclosed to indicate that you press a key on the keyboard.

. . .
: In examples, a horizontal ellipsis indicates one of the following possibilities:

- Additional optional arguments in a statement have been omitted.
- The preceding item or items can be repeated one or more times.
- Additional parameters, values, or other information can be entered.

⋮
: A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.

( )
: In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.

[ ]
: In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.

{ }
: In format descriptions, braces surround a required choice of options; you must choose one of the options listed.

boldface text
: Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.

italic text
: Italic text represents information that can vary in system messages (for example, Internal error *number*).

UPPERCASE TEXT

Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ).

UPPERCASE TEXT

Uppercase letters indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.

Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.

numbers

Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Introduction

The VMS DECwindows software provides a complete environment for developing and interacting with graphics-oriented applications. The DECwindows software presents a common network-transparent application programming environment for windowing, graphics, and user interface services. It is a single-appearance interface that is based upon the industry-standard X Window System, Version 11. The system comprises several components: a server, device drivers, the network protocol and transport mechanisms, and the DECwindows Xlib and Toolkit programming libraries.

1.1 About This Manual

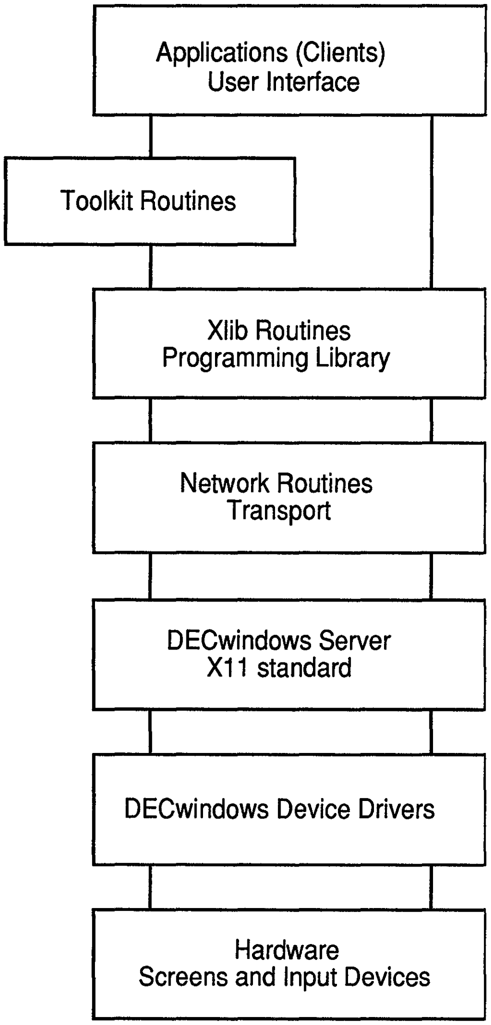
This document focuses on the device driver software that provides the DECwindows device interface. The document provides the necessary information for writing either a port driver or a class driver and for understanding the DECwindows device driver contents and concepts. Accessing the hardware directly is beyond the scope of this manual. You should refer to the hardware documentation for hardware information and to the *VMS Device Support Manual* for VMS device driver information concerning port/class driver programming and the related VMS data structures. It may be necessary to refer to the DECwindows data structures described in Appendix A while you read the chapters.

This chapter introduces the various drivers that make up the DECwindows device interface. A brief overview of the layers of DECwindows software that are above the device drivers is presented first.

1.2 DECwindows Architecture

The DECwindows architecture shown in Figure 1–1 identifies and illustrates the hierarchical layers of the DECwindows software from high-level programs of the user/application layer to the low-level programs of the device driver layer. As the figure illustrates, applications do not program to or call the drivers directly. All driver functions are available to application programs through the use of Xlib routines or DECwindows toolkit routines.

Figure 1–1 DECwindows Architecture



ZK-0089A-GE

Xlib is a library of more than 300 medium-level routines for creating and managing a window environment. The routines define the mapping of the X11 network protocol to a procedure library. Xlib provides a way for client applications to communicate with the DECwindows server without having to deal explicitly with the network protocol or server. Client applications can directly call the Xlib routines to manage DECwindows resources such as windows, color maps, input devices, and bitmap graphics services. Xlib then converts these routines into protocol requests that the transport sends to the server.

The DECwindows Toolkit is built on top of Xlib and provides convenient access to the Xlib features. The toolkit allows a programmer to use the power of the Xlib routines from a higher level of access. It streamlines the coding and saves time in the programming task.

The transport is a general or transparent data transfer mechanism within DECnet; it does not interpret or need to recognize the particular format of the data that it transfers. The transport operates symmetrically on both ends of the client/server connection in that it buffers and sends requests in the form of output data transfers to the server and sends input events, errors, and replies in the form of input data transfers to Xlib.

The server program is a lower-level component of the architecture that allows application interfaces to interact with all supported systems in the same way. The server converts the transport layer request to a command that can be executed by the appropriate device driver.

When the user of an application enters data, the server receives input from the device drivers and passes event packets back through the transport layers to Xlib and DECwindows Toolkit routines. The server supports asynchronous input to the application and asynchronous output from the application to the device.

1.3 DECwindows Device Drivers

DECwindows workstation device drivers are the lowest level of the DECwindows system, providing the device interface and support functions. The following are the family of VAX workstations that DECwindows currently supports:

- VAXstation II
- VAXstation II/GPX
- VAXstation 2000
- VAXstation 2000/GPX
- VAXstation 3000 series

The drivers support screens, keyboards, and system pointing devices. The server design and device driver support also allow nonstandard or "extension" input devices to be added. You can add nonstandard devices such as tablets and dial boxes[1] that require you to add your own input driver module to the driver software. Table 1–1 shows the relationship of the driver software modules to the various workstation families and hardware units. The table includes the device name used for each device in the VMS I/O database.

As shown in Table 1–1, IKDRIVER and IMDRIVER are class input drivers. IKDRIVER supports LK201 keyboard byte-stream processing and IMDRIVER supports mouse data input processing.

[1] A dial box is an analog control device having a set of knobs for variable adjustment to various graphic images and movements on the screen.

Table 1–1 Driver Software/Hardware Relationship

Class Input Driver	Hardware Unit	Device Name	Device
IKDRIVER	Pseudodevice	IKA0	LK201 keyboard
IMDRIVER	Pseudodevice	IMA0	VSXXX mouse or tablet
(Yourdriver)	Pseudodevice	Uxxx	(Yourdevice)

Port Input Driver	Hardware Unit/Controller	Device Name	System Type
YEDRIVER	Serial line 0	TTA0	VAXstation 2000 (monochrome and color)
	Serial line 1	TTA1	
GAADRIVER	Serial line 0	GAA1	VAXstation 3000 series and II/GPX
	Serial line 1	GAA2	
GCADRIVER	Serial line 0	GCA1	VAXstation II monochrome
	Serial line 1	GCA2	
DZDRIVER	DZQ11, DZV11	TT	VAXstation II, MicroVAX II
	DZ11, DZ32	TT	Large VAX systems
YFDRIVER	DHV11, DHU11	TX	MicroVAX II

Output Driver	Hardware Unit/Controller	Device Name	Workstation Type with VR260 Monitor
GABDRIVER	Busless CPU and GPX video controller	GAA0	VAXstation 2000/GPX
GCBDRIVER	Busless CPU and B/W video controller	GCA0	VAXstation 2000 (monochrome)
GAADRIVER[1]	Q22-bus CPU and VCB02 video controller	GAA0	VAXstation II/GPX, VAXstation 3000 series
GCADRIVER[1]	Q22-bus CPU and VCB01 video controller	GCA0	VAXstation II (monochrome)

Common Driver	Hardware Unit/Controller	Device Name	System Type
INDRIVER	All DECwindows driver/server interfaces	INA0	All DECwindows systems

[1] The output driver also contains the port input driver software.

The GxBDRIVERs are output drivers for the VAXstation 2000. GABDRIVER supports output data processing to a VAXstation 2000 color monitor; GCBDRIVER supports output to a VAXstation 2000 monochrome monitor.

The GxADRIVERs are output drivers for the VAXstation II and the VAXstation 3000 series. GAADRIVER supports output data processing to a VAXstation color monitor; GCADRIVER supports output to a VAXstation II monochrome monitor. Note that the port input driver component is built into these output drivers.

INDRIVER is the common DECwindows driver required for each workstation server interface.

YEDRIVER is the port input driver required for the VAXstation 2000 input devices. DZDRIVER and YFDRIVER are only used for nonstandard workstation input devices. These port drivers are not described in this manual.

The modular DECwindows architecture allows for expansion, utilizing other device-specific driver extensions, including ones not furnished by DIGITAL. However, you cannot add a driver for input devices other than a keyboard or system pointing device without a server extension. Because server and Xlib extensions are not implemented in this release, such a driver cannot be added. You can replace an existing driver with a new one. For example, you can replace a class input mouse driver with one for a tablet.

1.3.1 Features Supported by the Device Drivers

VMS drivers supplied with DECwindows software provide the following functionality:

- Keyboard, pointer, and button input
- Color services
- Monochrome frame buffer system
- Cursor services
- Device characteristics information
- Input queue
- Tablet input (as a system pointing device)
- Graphics output
- Multiscreen support
- Pointer acceleration control
- Mouse motion event prebuffering and compression
- Keyboard pseudomouse

All window management is performed by the server, therefore there are no window services within the driver. The drivers treat the physical screen as a single rectangular bitmap.

Keyboard input is supplied in the form of raw LK201 scan codes. According to the X11 standard protocol, translation services are available using Xlib routines. Key autorepeat is simulated by the drivers. The LK201 keys are set in up/down transition detection mode. The drivers support the pseudomouse feature where the keyboard can simulate the mouse functions in the event of mouse failure. Pointer acceleration can be controlled by calls to a $QIO system service and X11 type acceleration table in the driver. The driver provides mouse motion event prebuffering and compression for improved motion event system response. These features are selectable by the server using the $QIO interface.

The drivers also provide multiscreen support interfacing a single input device with multiple output devices. Screens of multiple DECwindows workstations can be attached to a single pointing device. The pointer can move off the top of one screen into the bottom of another, or off screen to the right or left into another. After a screen saver timeout, all screens come alive with any mouse movement or keystroke.

1.4 Driver Architecture

A DECwindows workstation device driver is divided into multiple driver modules. The following lists the modules that make up a DECwindows driver:

- Class input driver
- Port input driver
- Output driver
- Common driver

The division of the DECwindows driver into various modules provides flexibility and ease of coding in terms of driver development and ease of upgrade for the varied workstation types and devices. The various driver modules communicate by means of vector tables and shared data in the unit control block.

The architecture of the software modules or basic subsystems for workstation families with busless CPUs is shown in Figure 1–2. Input drivers process data from the keyboard or mouse and pass it to the server. The input drivers also process output data, such as keyboard LED information, from the server to the keyboard. An output driver processes graphics and windowing requests in the form of output data that pass from the server to the monitor.

Workstation families with Q22 bus-based CPUs and VCB01/VCB02 video controllers use the GxADRIVER modules. For these, the driver architecture differs slightly in that the port input driver software is part of the output driver module, as shown in Figure 1–3. However, the port/class input characteristic of a DECwindows device interface remains the same.

Figure 1–2 DECwindows Driver Architecture for Busless CPUs

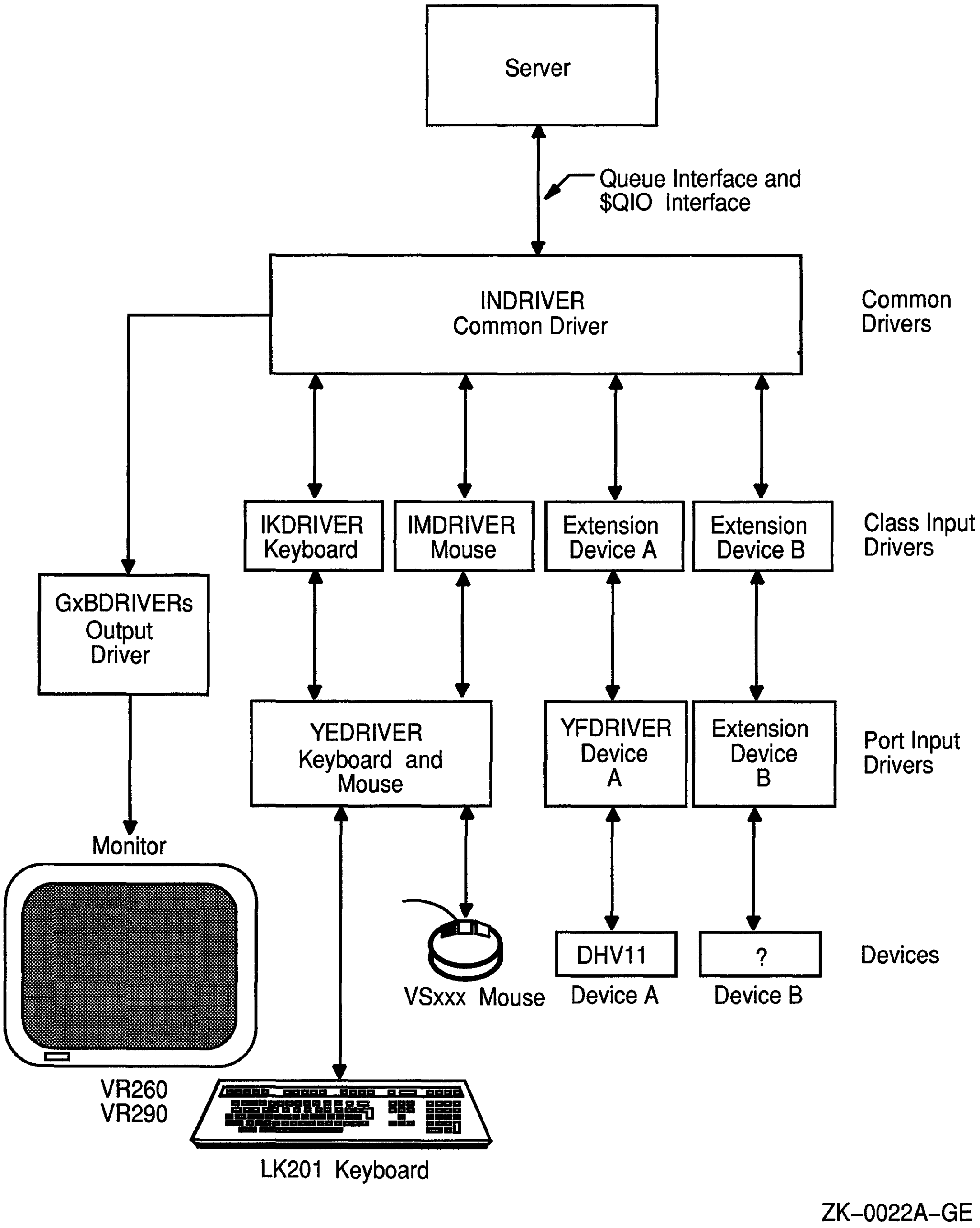
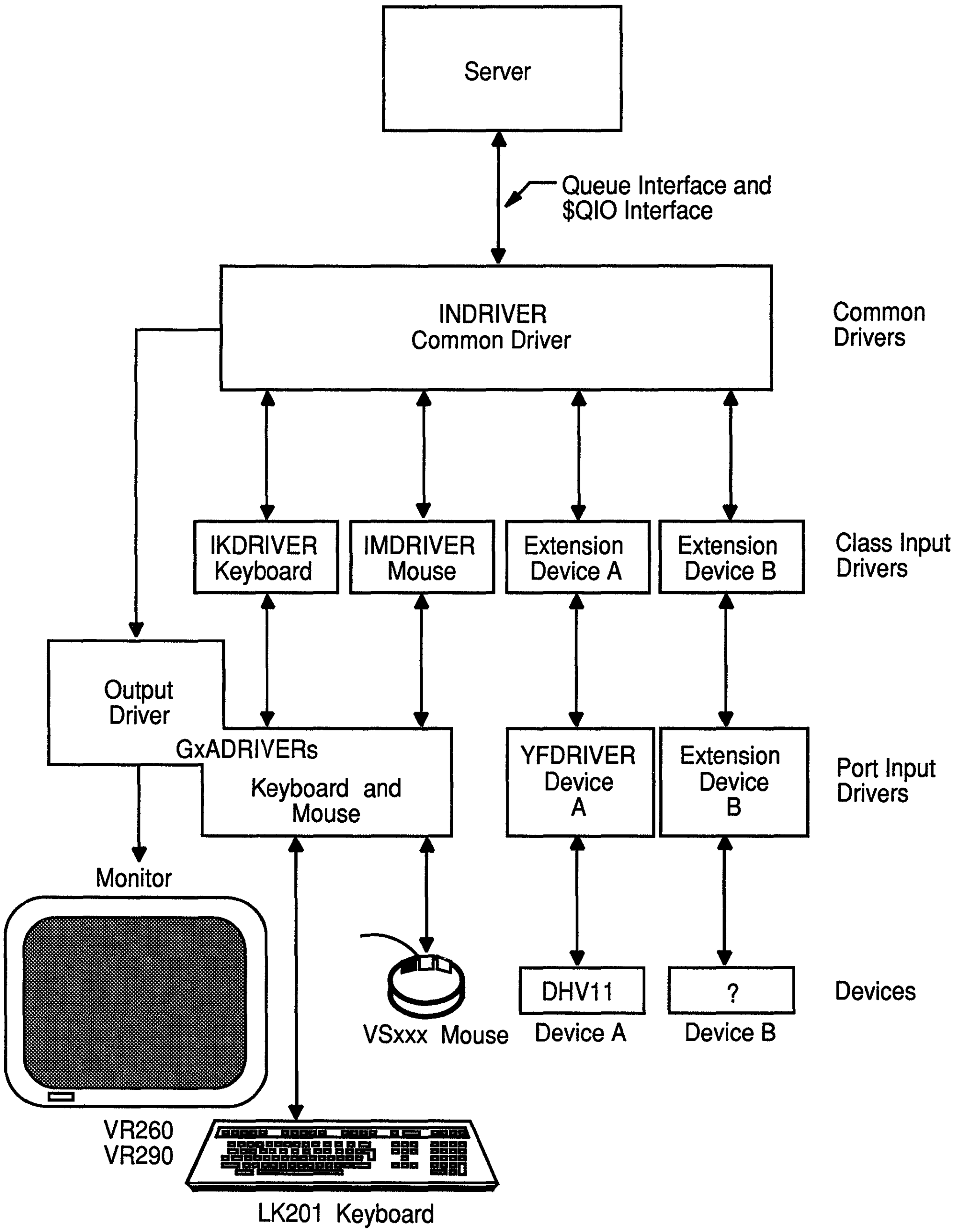


Figure 1–3 DECwindows Driver Architecture for Q22-Bus CPUs



ZK–0023A–GE

1.4.1 Driver/Server Interface

As shown in Figure 1–2 and Figure 1–3, the DECwindows driver presents two interfaces to the server; a queue interface and a $QIO interface.

Queue interface is an input event queue in memory that is shared between the server and drivers. This queue is interlocked for correct operation on multiprocessor workstations. The input queue receives all input events as they are generated by the drivers. A driver timestamps each input event as it inserts the event on the queue. Asynchronous input events supported by the drivers include the following:

- Key presses and releases
- Mouse movement
- Mouse button presses and releases

The use of a single queue ensures that all input is correctly time ordered when the server reads it.

$QIO interface is the second driver/server interface, which is also common to VMS device drivers. Generally only the DECwindows server should make $QIO calls or access the queue. Applications should use the DECwindows library (Xlib) routines to perform these functions.

1.4.2 Common Driver Function

The common driver (INDRIVER) is the interface between an input or an output driver and the DECwindows server. It monitors the input queue and supports the server protocol of the interface. The common driver handles device-independent processing and functions that are common to all workstations. Common driver $QIO service routines support $QIO calls from the server. Getting device information or parsing $QIO parameters for all the device drivers are examples of the common driver function.

1.4.3 Port/Class Input Driver Function

The **input drivers** handle input data transfers from the keyboard, mouse, tablet and other input devices to the common driver and on to the server. They are developed based on a modular **port/class driver** interface designed for VMS that allows for new input devices or serial line hardware to be easily added. Port/class input drivers are bound together by means of the system UCB data structure and form a port/class interface. Note that parts of the full VMS terminal port/class software are not used by DECwindows. The VMS terminal port/class software is described in the *VMS Device Support Manual.*

The **port drivers** provided by VMS receive interrupts from and transmit data to the hardware ports. Port drivers service input serial lines only. These serial lines may be part of the graphics hardware (VCB02 controller) or a standard serial line controller (VAXstation 2000 DZ controller). Data received by the port driver is passed to the appropriate class driver for interpretation.

The **class drivers** provided by VMS include the keyboard driver (IKDRIVER) and the mouse driver (IMDRIVER). A class driver interprets a byte stream from an input device and then formats the data into an event packet for the input queue. Because each input device, such as a keyboard or tablet, must interpret a different byte stream protocol, there is one class driver per input device type on a system.

1.4.4 Output Driver Function

The output driver processes graphics and windowing requests from the server to the screen. Output drivers manage the output functions of the video controller. For example, the output driver performs all device-dependent processing, such as receiving device interrupts, manipulating the color map, drawing, and managing the current state of the graphics hardware. Note that the output modules (GxADRIVERs) also contain port input driver software, as shown in Figure 1–3.

In addition to the input queue, some DECwindows video devices use an output queue. For instance, drivers for color devices support an output queue, while monochrome drivers do not. This queue is the interface for drawing operations from the server (or applications). Like the input queue, the output queue is in nonpaged pool shared by the driver and the server. Drawing packets are inserted into the queue by the server. The driver removes the packets from the queue and executes them in the queued order.

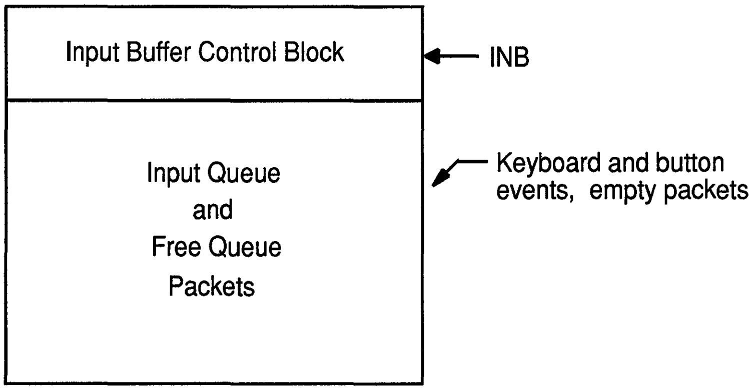
2 Common Driver/Server Interface

The common driver/server link is made up of two interface types; the common input queue interface and the common $QIO interface. The main DECwindows device driver interface to the server is a buffer containing a queue of event packets formatted for the X11 standard protocol. This chapter describes the buffer/input queue and the protocol of the driver/server interface. Also described is the $QIO interface. The service mechanism that supports the $QIO calls is described in Chapter 5. Data structures referenced in this chapter are described in detail in Appendix A.

2.1 Driver/Server Common Buffer

The common driver manages an input buffer that the driver shares with the server. Figure 2–1 illustrates the input buffer structure with its queues. The shared input buffer is a block allocated in nonpaged pool. It contains a control block or header and two queues: an input event queue and a free queue. The queues are self-relative interlocked queues that provide an efficient communication path for frequent driver/server operations. Using the input buffer control block (INB), the common driver monitors each queue containing input event packets (INPs). Each packet stored in the buffer contains a forward and a backward pointer (FLINK and BLINK) to complete the event chain that defines each self-relative queue.

Figure 2–1 Input Buffer General Structure



ZK–0026A–GE

2.1.1 Input Queue and Motion History Buffer

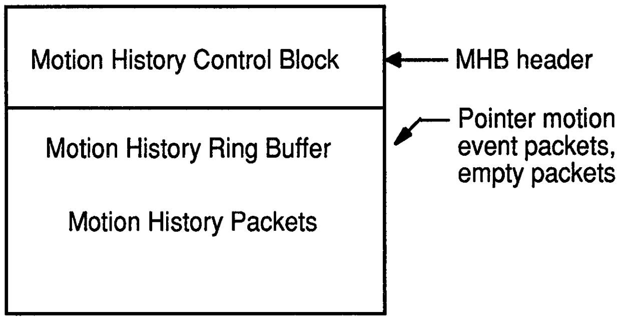
The server may create a pointer motion history buffer (MHB) to improve system response to pointer movement. The input queue always maintains the most recent motion events along with other input device events for the server. However, if the motion compression feature is enabled, the server may not receive all of the motion events generated in the input queue. If the server requires motion events that were not delivered because of motion compression, the server can access the motion history buffer for the older motion events.

Like the INB, the MHB is allocated in nonpaged pool. The server issues an Initialize Motion History $QIO call (described in this chapter) specifying the desired size in pages. The ring buffer (shown in Figure 2–2) contains a control block or header in the first 16 bytes, followed by a ring of 8-byte motion history packets (MHPs) throughout the remaining allocated space. Like the input buffer, the motion history buffer contains active event packets and free packets. When a server/driver searches the queue, the oldest motion history event packet and free packet are located in the ring with *put* and *get* pointers in the MHB header. Each motion history packet contains the *x* and *y* movement with an event timestamp. Refer to Appendix A for detailed field information.

Once an MHB is created, a pointer motion event is first stored in a motion history packet in the MHB and then copied into an input packet in the input buffer shared with the server. However, the server may disable the MHB by setting the INB$V_MHB_BUSY bit, which forces the buffering of all events by way of the input buffer only.

When motion event compression is enabled (by the Set Motion Compression $QIO), the motion event decoder removes the oldest motion event packet from the input queue as it inserts the newest event. Thus, the removed events (oldest of a large burst of pointer motion and/or those not yet retrieved by the server) are lost, yielding motion compression. The number of lost motion events or motion compression hits is stored in counter DWI$L_PTR_MOTION_COMP_HIT. However, if necessary, the server program can recover lost events by accessing the motion history buffer, instead of the input queue from which they are missing.

Figure 2–2 Motion History Buffer General Structure


Motion History Control Block
MHB header
Motion History Ring Buffer
Motion History Packets
Pointer motion event packets, empty packets

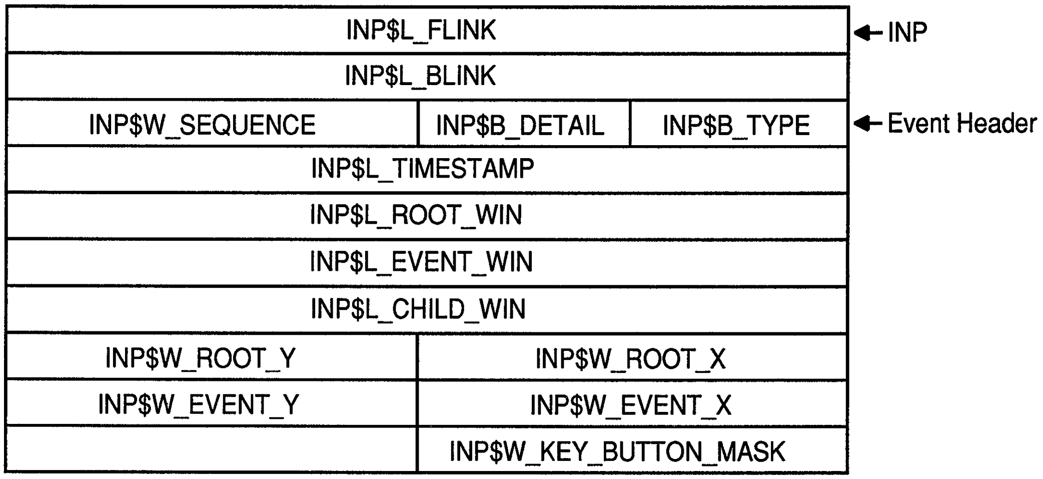
ZK–0167A–GE

2.1.2 Input Queue Event Packet

The input packet structure (INP) in the input queue defines the packet format used in the interface between the device driver and the DECwindows server. The basic DECwindows format of the input packet, shown in Figure 2–3, is compatible with the X event in the X Window System protocol.

Depending on the driver, some fields in the input packet of certain events may vary. The packet illustrated in Figure 2–3 is a typical keyboard- or mouse-generated input event for key/button transitions and mouse motion. The first 12 bytes (3 longwords) are common to all event types (see Figure 2–3). The event information is always 32 bytes long, excluding the forward/backward pointers (FLINK/BLINK). The FLINK and BLINK pointers link (in proper order) all the event packets of the input queue. Refer to Appendix A for detailed field information.

Figure 2–3 Queue Event Packet Format



INP$L_FLINK			← INP
INP$L_BLINK			
INP$W_SEQUENCE	INP$B_DETAIL	INP$B_TYPE	← Event Header
INP$L_TIMESTAMP			
INP$L_ROOT_WIN			
INP$L_EVENT_WIN			
INP$L_CHILD_WIN			
INP$W_ROOT_Y	INP$W_ROOT_X		
INP$W_EVENT_Y	INP$W_EVENT_X		
	INP$W_KEY_BUTTON_MASK		

ZK–0088A–GE

2.1.3 Queue Processing of Input

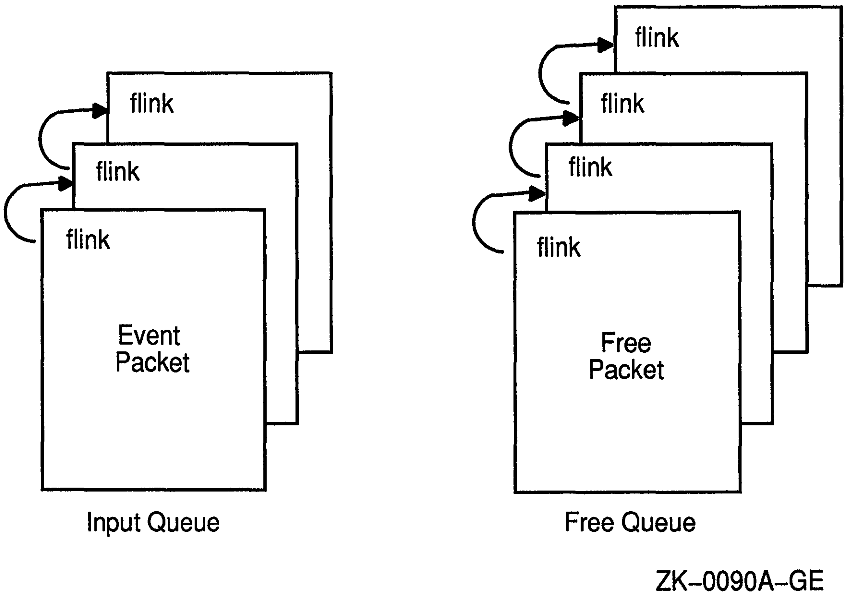
There is one input queue and one free queue for each keyboard/mouse pair for input. As events occur in the device, the class driver gets a free packet from the free queue and inserts it into the input queue. The class driver links all active keyboard and pointer event packets in the input queue using the forward link (FLINK) and backward link (BLINK) INP fields (see Figure 2–4).

Because there is a single input queue shared by the input devices, packet-link pointers ensure that all events are correctly time ordered when they are read by the server. The size of the input queue varies inversely with the size of the free queue; as more packets move to the input queue, the number of free packets diminishes.

The common driver checks the queue at timed intervals to see if there is input. During the hardware vertical retrace interval (VSYNC) the driver checks the INPUT_QUEUE_FLINK pointer in the input buffer control block (INB). If the queue is not empty or the queue is not being accessed by the server, the driver wakes the server to signal the presence of input.

When the server responds, the server processes the event data, removes the event packet from the input queue, and inserts the packet on the free queue. The server processes each event packet in the queue until the queue is empty.

Figure 2–4 Input Queue and Free Queue



2.2 $QIO Common Interface

The INDRIVER module contains function decision table (FDT) routines that make up a $QIO common interface. The $QIO common interface provides for initialization and information requests from the server or server extension to a device. The $QIO interface is used for infrequent operations that do not require synchronization with input or output requests or when notification upon completion of a request is needed.

2.3 $QIO Calls to DECwindows Drivers

This section presents the output $QIO calls used in a server that are supported by services within the DECwindows common driver. The $QIO system service format is presented first.

$QIO calls must be issued to a physical device, as they cannot be directed to a pseudodevice (such as IKA0 for the keyboard decoder). Initially, using the $ASSIGN system service, the appropriate device name is assigned to an I/O channel. The channel number entry is required for the **chan** parameter in the $QIO system service call. Physical device names on a GPX workstation are GAA0 for output to the screen, GAA1 for the keyboard, and GAA2 for the mouse (see Table 1–1). Note that the DECwindows environment provides logical names (DECW$WORKSTATION, DECW$KEYBOARD, and DECW$POINTER) to point to the physical device.

$QIO System Service

The Queue I/O Request system service queues an I/O request to a channel associated with a device. The SYS$QIO format described next applies to all the $QIO calls presented in this chapter. For more information on SYS$QIO refer to *VMS System Services Reference Manual.*

FORMAT

SYS$QIO *[efn],chan,func,[iosb],[astadr],[astprm] ,p1,p2,p3[,p4][,p5][,p6]*

arguments

efn is the event flag number of the I/O operation. The **efn** argument is a longword containing the number of the event flag.

chan is the I/O channel assigned ($ASSIGN) to the device name to which the request is directed. The **chan** argument is a longword containing the number of the I/O channel; however, $QIO uses only the low-order word.

func is the device-specific function code specifying the operation to be performed. The **func** argument is a longword containing the function code.

iosb is the I/O status block to receive the final completion status of the I/O operation. The **iosb** argument is the address of the quadword I/O status block.

astadr is the AST service routine to be executed when the I/O completes. The **astadr** argument is the address of a longword that is the entry mask to the AST routine.

astprm is the AST parameter to be passed to the AST service routine. The **astprm** argument is a longword containing the AST parameter.

pl is the function modifier specifying the service being called within the basic function code (IO$K_DECW_xxx).

p2 to **p6** are the function-specific parameters being passed.

2.4 Sense Mode Calls

The FDT sense mode routines within the common driver service the $QIO sense mode function calls from a server. The following sense mode calls are supported by the DECwindows common driver.

- Sense Keyboard Information
- Sense Keyboard LED
- Sense Motion Compression
- Sense Operator Window Key
- Sense Pointer Acceleration
- Sense Pseudomouse Key
- Sense Screen Saver
- Get Device Information

This section defines the specific argument data required for each $QIO call within the sense mode functions serviced by the common driver. Each of these calls requires the IO$_SENSEMODE function code.

Sense Keyboard Information

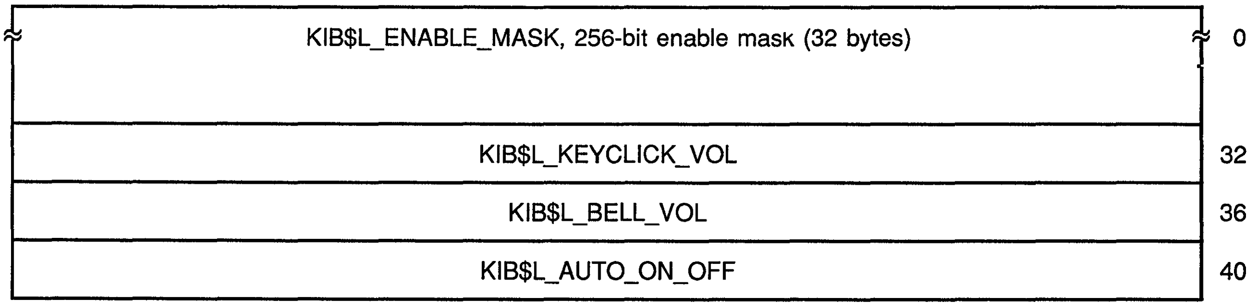
The Sense Keyboard Information $QIO function returns the current functional characteristics or information concerning the keyboard device. Table 2–1 provides the argument information required for the Sense Keyboard Information $QIO call.

Figure 2–5 and Table 2–2 show and define the data structure that passes the keyboard information requested for the $QIO call.

Table 2–1 Argument Data for Sense Keyboard Information $QIO Call

$QIO Argument	Required Data
func	IO$_SENSEMODE function code.
p1	IO$K_DECW_KB_INFO function modifier.
p2	Address of the keyboard information (characteristics) block (KIB).
p3	Address of the longword that stores the length of the keyboard information block.
p4, p5, p6	Set to 0.

Figure 2–5 Keyboard Information Block



KIB$L_ENABLE_MASK, 256-bit enable mask (32 bytes)	0
KIB$L_KEYCLICK_VOL	32
KIB$L_BELL_VOL	36
KIB$L_AUTO_ON_OFF	40

Table 2–2 Keyboard Information Block Fields

Field Name	Contents
KIB$L_ENABLE_MASK	Entry to the 256-bit autorepeat enable mask for the LK201 keys. The mask defines which keys are in autorepeat mode. The bits are numbered 0 through 255 and each bit position corresponds directly to a specific key position on an LK201 keyboard. For example, using decimal keycode numbering, mask-bit 90 corresponds to the 90 key position (F5) on the LK201 keyboard.
KIB$L_KEYCLICK_VOL	A longword specifying the current keyclick volume in percent. A value of 100 indicates the loudest click while a 0 indicates the click is off. A value of –1 indicates that a default value of 70 percent volume is set.
KIB$L_BELL_VOL	A longword specifying the current bell volume in percent. A value of 100 indicates the loudest ring is set while a 0 indicates the bell is off. A value of –1 indicates that a default volume of 70 percent is set.
KIB$L_AUTO_ON_OFF	A value of 0 that indicates the autorepeat feature is disabled for all keys. A value of 1 indicates that autorepeat is enabled for the keys specified (bits set) in the KIB$L_ENABLE_MASK.

Sense Keyboard LED

The Sense Keyboard LED $QIO function gets the current status of the keyboard LEDs. The target device is the keyboard. Table 2–3 provides the argument information required for the Sense Keyboard LED $QIO call.

Table 2–3 Argument Data for Sense Keyboard LED $QIO Call

$QIO Argument	Required Data	
func	IO$_SENSEMODE function code.	
p1	IO$K_DECW_KB_LED function modifier.	
p3	Address of the longword keyboard status mask (DWI$L_KB_LIGHTS). Mask bits correspond to the keyboard LEDs as follows:	
	DECW$M_LIGHT1	Wait
	DECW$M_LIGHT2	Compose
	DECW$M_LIGHT3	Lock
	DECW$M_LIGHT4	Hold Screen
	A mask bit is 1 when the LED is on, or 0 when the LED is off.	
p2, p4, p5, p6	Set to 0.	

Sense Motion Compression

The Sense Motion Compression $QIO function gets the status of a pointing device's motion compression mode. The $QIO returns the motion compression flag bit status in **p2**. The target is the pointing device. Table 2–4 provides the argument information required for the Sense Motion Compression $QIO call.

Table 2–4 Argument Data for Sense Motion Compression $QIO Call

$QIO Argument	Required Data
func	IO$_SENSEMODE function code.
p1	IO$K_DECW_MOTION_COMP function modifier.
p2	Address of the motion compression state longword. The longword contains 1 when motion compression is on, or 0 when motion compression is off.
p3, p4, p5, p6	Set to 0.

Sense Pointer Acceleration

The Sense Pointer Acceleration $QIO function gets the state of the pointer acceleration table and threshold used by the acceleration routine. The $QIO returns the addresses of the pointer acceleration values from the pointer input UCB extension (DWI$W_PTR_ACCEL_NUM, DWI$W_PTR_ACCEL_DEN, and DWI$W_PTR_ACCEL_THR, in **p2**, **p3**, and **p4**, respectively). The target is the pointing device. Table 2–5 provides the argument information required for the Sense Pointer Acceleration $QIO call.

Table 2–5 Argument Data for Sense Pointer Acceleration $QIO Call

$QIO Argument	Required Data
func	IO$_SENSEMODE function code.
p1	IO$K_DECW_PTR_ACCEL function modifier.
p2	Address of the word that stores the pointer acceleration numerator (DWI$W_PTR_ACCEL_NUM).
p3	Address of the word that stores the pointer acceleration denominator (DWI$W_PTR_ACCEL_DEN).
p4	Address of the word that stores the pointer acceleration threshold (DWI$W_PTR_ACCEL_THR).
p5, p6	Set to 0.

Sense Pseudomouse Key

The Sense Pseudomouse Key $QIO function gets the key code information that invokes the pseudomouse mode. The $QIO returns the selection key and selection key modifier codes in use from the keyboard input UCB extension (DWI$B_KB_PMOUSE_KEY and DWI$B_KB_PMOUSE_MOD). The target device is the keyboard. Table 2–6 provides the argument information required for the Sense Pseudomouse Key $QIO call.

Table 2–6 Argument Data for Sense Pseudomouse Key $QIO Call

$QIO Argument	Required Data
func	IO$_SENSEMODE function code.
p1	IO$K_DECW_PMOUSE_KEY function modifier.
p2	Address of the byte containing the LK201 key code that selects the pseudomouse mode.
p3	Address of the longword that contains the pseudomouse selection modifier key code.
p4, p5, p6	Set to 0.

Sense Operator Window Key

The Sense Operator Window Key $QIO function finds the key code that invokes the operator window. The target device is the keyboard. Table 2–7 provides the argument information required for the Sense Operator Window Key $QIO call.

Table 2–7 Argument Data for Sense Operator Window Key $QIO Call

$QIO Argument	Required Data
func	IO$_SENSEMODE function code.
p1	IO$K_DECW_OPWIN_KEY function modifier.
p2	Address of the byte containing the LK201 key code that selects the operator window.
p3	Address of the longword mask that specifies whether a control or shift key is used as a modifier in the selection of the operator window mode.
p4, p5, p6	Set to 0.

Sense Screen Saver Timeout

The Sense Screen Saver Timeout $QIO function gets the current screen saver timeout value in seconds. The target device is the output monitor. Table 2–8 provides the required argument information for the Set Screen Saver Timeout $QIO call.

Table 2–8 Argument Data for Sense Screen Save Timeout $QIO Call

$QIO Argument	Required Data
func	IO$_SENSEMODE function code.
p1	IO$K_DECW_SCRSAV function modifier.
p2	Address of the longword specifying the current timeout in seconds. A value of 0 indicates that the screen saver function is disabled.
p3, p4, p5, p6	Set to 0.

Get Device Information

The Get Device Information $QIO sense-mode function returns the address and size of the device information block (DVI). The target device is the output display. The function returns a pointer to the (read-only) DVI block. The DVI contains static device information such as the size of the memory frame buffer, the resolution of the screen, the number of bits per pixel in the frame buffer, the number of cursor planes and the width and height of the cursor bitmap. See the DVI block in Figure A–1 and Table A–1 for more detailed field information. Table 2–9 provides the required argument information for the Get Device Information $QIO call.

Table 2–9 Argument Data for Device Information $QIO Call

$QIO Argument	Required Data
func	IO$_SENSEMODE function code.
p1	IO$K_DECW_DEVICE_INFO function modifier.
p2	Address that stores the DVI block address.
p3	Address that stores the DVI block length.
p4, p5, p6	Set to 0.

2.5 Set Mode Calls

The FDT set-mode routines within the common driver service the $QIO function calls from a server that set various device characteristics. The following set-mode calls are supported by the DECwindows common driver:

- Enable Input
- Initialize Motion Buffer
- Set Attach Screen
- Set Cursor Pattern
- Set Cursor Position
- Set Keyboard Information
- Set Keyboard LED
- Set Motion Compression
- Set Operator Window Key
- Set Pointer Acceleration
- Set Pseudomouse Key
- Set Screen Saver
- Ring Keyboard Bell

This section defines the specific argument data required for each $QIO call within the set-mode functions serviced by the common driver. Each of these calls requires the IO$_SETMODE function code.

Enable Input

The Enable Input $QIO set-mode function creates a nonpaged shared memory buffer for communication between the driver and the caller. This $QIO function returns the page frame numbers (PFNs) of the buffer. The caller then calls the $CRMPSC system service to map the PFNs (PFNMAP) into its process address space. The target device is the keyboard. However, all input devices share the input queue within the buffer (INB). If this is not the first request to build the buffer, the $QIO function just returns the data for the buffer previously allocated.

The size parameter (**p2**) is an input/output parameter. The caller requests a buffer size and the common driver returns the size in pages (INB$L_PFN_COUNT). The input queue and free lists are initialized on the first invocation. Table 2–10 provides the required argument information for the Enable Input $QIO call.

Table 2–10 Argument Data for Enable Input $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_ENABLE_INPUT function modifier.
p2	The size (number of pages) of the input buffer.
p3	Address of the array of PFN longwords.
p4, p5, p6	Set to 0.

Initialize Motion Buffer

The Initialize Motion Buffer $QIO function creates the motion history buffer (MHB) data structure, initializes its fields, and returns the buffer size in pages. Pointing device motion events are stored in 8-byte packets (MHPs); thus the buffer event capacity is determined as follows:

```
N events = (N buffer_pages * 512 - 16 byte MHB header)/ 8
```

The number of pages for the buffer is specified in the **p2** parameter of the $QIO call. The address of the number of pages allocated is returned in **p2**. By default, the motion history buffer is disabled when DECwindows starts up. Table 2–11 provides the argument information required for the Initialize Motion Buffer $QIO call.

Table 2–11 Argument Data for Initialize Motion Buffer $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_MOTION_BUFFER_INIT function modifier.
p2	Size of the motion history buffer in pages.
p3, p4, p5, p6	Set to 0.

Set Attach Screen

The Set Attach Screen $QIO set-mode function attaches the target screen to an input device or to another screen. The specific attach-screen function is selected with parameter **p2** (see Table 2–12). All attached screens work with the same input pointing device. The $QIO system service also tests the output device being called to ensure that it is a DECwindows workstation type. Table 2–12 provides the required argument information for the Set Attach Screen $QIO call. The attach-screen function has the following conditional characteristics and features:

- If more than one screen is attached to the single pointing device, one pointer is shared among the screens. Positioning the pointer on one screen removes it from all the others.
- If two screens are not attached to the same input pointing device, the screens cannot be attached to one another.
- As a screen is attached to an input pointing device, it is detached from all other screens.

Table 2–12 Argument Data for Set Attach Screen $QIO Call

$QIO Arg	Required Data	
func	IO$_SETMODE function code.	
p1	IO$K_DECW_ATTACH_SCREEN function modifier.	
p2	The specific function requested in the options of the attach-screen submodifier group.	
	IO$K_DECW_AS_TO_INPUT	Attaches screen to pointing device
	IO$K_DECW_AS_TO_RIGHT	Attaches screen to the right of the active screen
	IO$K_DECW_AS_TO_LEFT	Attaches screen to the left of the active screen
	IO$K_DECW_AS_TO_TOP	Attaches screen to the top of the active screen
	IO$K_DECW_AS_TO_BOTTOM	Attaches screen to the bottom of the active screen
p3	A string descriptor naming the device being connected to.	
p4	A number passed back to the server in the INP$L_ROOT_WIN field of the input packet.	
p5, p6	Set to 0.	

Set Cursor Pattern

The Set Cursor Pattern $QIO set-mode function sets the cursor (pointer) pattern.

Much of the cursor information is stored in the UCB common output extension (DECW). The bitmap image length differs in single-plane and multiplane cursor systems. The caller must check the system type and pass the correct image length. To obtain the address of the DVI block where the number of cursor planes is stored, use the Get Device Information $QIO call.

The style (**p5**) is a longword that specifies how the cursor is presented against the background screen. Possible values are

0	dynamic NAND
1	dynamic OR
2	NAND
3	OR

The **p5** parameter is not used for multiplane cursor systems. Table 2–13 provides the required argument information for the Set Cursor Pattern $QIO call.

Table 2–13 Argument Data for Set Cursor Pattern $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_CURSOR_PATTERN function modifier.
p2	Address of the bitmap image.
p3	Length (number of words) of the bitmap.
p4	A longword defining the hotspot *x*-, *y*-coordinates.
p5	Defines the cursor display style.
p6	Set to 0.

Set Cursor Position

The Set Cursor Position $QIO set-mode function sets the cursor *x* and *y* position. It returns a SS$_BADPARAM message if the cursor (pointer) is out of range along the *x*- and *y*-axes. If multiple screens are attached to a single mouse, it removes the mouse from all other attached screens and places the pointer on the home screen. The target device is the output display. The cursor information is stored in the UCB common output extension (DECW). Table 2–14 provides the required argument information for the Set Cursor Position $QIO call.

Table 2–14 Argument Data for Set Cursor Position $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_CURSOR_POSITION function modifier.
p2	Defines the *x* position of the cursor on the screen.
p3	Defines the *y* position of the cursor on the screen.
p4, p5, p6	Set to 0.

Set Keyboard Information

The Set Keyboard Information $QIO set-mode function enables/disables keyboard functions and sets various functional characteristics of the keyboard device. Table 2–15 provides the argument information required for the Set Keyboard Information $QIO call.

Figure 2–6 and Table 2–16 show and define the data structure that passes required keyboard information for the $QIO call.

Table 2–15 Argument Data for Set Keyboard Information $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_KB_INFO function modifier is used to set or adjust the keyclick and bell volume and to set the alphanumeric keys (main keyboard) in up/down transition mode of event reporting. An IO$M_DECW_KEYCLICK optional function modifier is used to set the keyclick volume, IO$M_DECW_BELL is used to set the bell volume, and IO$M_DECW_AUTOREPEAT is used to enable/disable the autorepeat feature.
p2	Address of the keyboard information (characteristics) block (KIB).
p3	Specifies the length of the keyboard information block (KIB$S_KBD_INFO).
p4	Defines the up/down or down-only transition mode of the alphanumeric keys. A value of 1 sets the FLAG$V_MAIN_KB_UPDOWN bit in the UCB keyboard input extension, enabling the up/down mode. A value of 0 selects the down-only transition mode for alphanumeric key events.
p5, p6	Set to 0.

Figure 2–6 Keyboard Information Block

KIB$L_ENABLE_MASK, 256-bit enable mask (32 bytes)	0
KIB$L_KEYCLICK_VOL	32
KIB$L_BELL_VOL	36
KIB$L_AUTO_ON_OFF	40

Table 2–16 Keyboard Information Block Fields

Field Name	Contents
KIB$L_ENABLE_MASK	Entry to the 256-bit autorepeat enable mask for the LK201 keys. The mask defines which keys are in autorepeat mode. The bits are numbered 0 through 255 and each bit position corresponds directly to a specific key position on an LK201 keyboard. For example, using decimal keycode numbering, mask-bit 90 corresponds to the 90 key position (F5) on the LK201 keyboard.
KIB$L_KEYCLICK_VOL	A longword specifying the keyclick volume in percent. A value of 100 specifies the loudest click while a 0 turns the click off. A value of –1 provides a default volume of 70 percent.
KIB$L_BELL_VOL	A longword specifying the bell volume in percent. A value of 100 specifies the loudest ring while a 0 turns the bell off. A value of –1 provides a default volume of 70 percent.
KIB$L_AUTO_ON_OFF	A value of 0 disables the autorepeat feature for all keys. A value of 1 enables autorepeat for the keys specified (bits set) in the KIB$L_ENABLE_MASK.

Set Keyboard LED

The Set Keyboard LED $QIO function sets the state of the keyboard LEDs. The target device is the keyboard. Table 2–17 provides the argument information required for the Set Keyboard LED $QIO call.

Table 2–17 Argument Data for Set Keyboard LED State $QIO Call

$QIO Argument	Required Data	
func	IO$_SETMODE function code.	
p1	IO$K_DECW_KB_LED function modifier.	
p2	A value of 1 turns the **p3** LEDs on, or a value of 0 turns the **p3** LEDs off.	
p3	A longword LED mask to set the keyboard lights to the state specified in **p2**. The mask bits correspond to the keyboard LEDs as follows:	
	DECW$M_LIGHT1	Wait
	DECW$M_LIGHT2	Compose
	DECW$M_LIGHT3	Lock
	DECW$M_LIGHT4	Hold Screen
p4, p5, p6	Set to 0.	

Note: The keyboard class driver ignores this $QIO for the Wait and Lock LEDs. The lock LED corresponds to the lock key on the LK201 keyboard and the wait LED identifies the keyboard pseudomouse mode.

Set Motion Compression

The Set Motion Compression $QIO function sets or clears the pointing device's motion compression mode. A value of 1 in the **p2** parameter turns on the compression mode, a value of 0 turns it off. The motion compression bit (FLAG$V_MOTION_COMP) is located in DWI$L_PTR_CTRL of the input UCB extension. The target is the pointing device. Table 2–18 provides the argument information required for the Set Motion Compression $QIO call.

Table 2–18 Argument Data for Set Motion Compression $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_MOTION_COMP function modifier.
p2	A value of 1 sets motion compression on, or a value of 0 turns motion compression off.
p3, p4, p5, p6	Set to 0.

Set Operator Window Key

The Set Operator Window Key $QIO set-mode function sets the key code that invokes the operator window. If the key code **p2** is not set, CTRL/F2 is the operator key default. The target device is the keyboard. Table 2–19 provides the required argument information for the Set Operator Window Key $QIO call.

Table 2–19 Argument Data for Set Operator Window Key $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_OPWIN_KEY function modifier.
p2	Byte specifying the LK201 key code that invokes the operator window. If set to 0, **p3** is ignored and CTRL/F2 is established as the default.
p3	Longword mask that specifies either a shift or control selection modifier key (INP$M_CONTROLMASK or INP$M_SHIFTMASK). The default selectors for the operator window mode are CTRL/F2. If a modifier is not used, **p3** must be 0.
p4, p5, p6	Set to 0.

Set Pointer Acceleration

The Set Pointer Acceleration $QIO function sets the pointer acceleration table states and the acceleration threshold value. The **p2**, **p3**, and **p4** parameters set states DWI$W_PTR_ACCEL_NUM, DWI$W_PTR_ACCEL_DEN, and DWI$W_PTR_ACCEL_THR, respectively, in the pointer input UCB extension. The target is the pointer device. Table 2–20 provides the argument information required for the Set Pointer Acceleration $QIO call.

Table 2–20 Argument Data for Set Pointer Acceleration $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_PTR_ACCEL function modifier.
p2	A word containing the pointer acceleration numerator for DWI$W_PTR_ACCEL_NUM and the acceleration routine.
p3	A word containing the pointer acceleration denominator for DWI$W_PTR_ACCEL_DEN and the acceleration routine. The default is 1.
p4	A word containing the pointer acceleration threshold for DWI$W_PTR_ACCEL_THR and the acceleration routine.
p5, p6	Set to 0.

Set Pseudomouse Key

The Set Pseudomouse Key $QIO function sets the key code to select the pseudomouse mode. The $QIO sets both the key and the key modifier scan codes in the keyboard input UCB extension (DWI$B_KB_PMOUSE_KEY and DWI$B_KB_PMOUSE_MOD). The default is CTRL/F3. The target device is the keyboard. Table 2–21 provides the argument information required for the Set Pseudomouse Key $QIO call.

Table 2–21 Argument Data for Set Pseudomouse Key $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_PMOUSE_KEY function modifier.
p2	The select pseudomouse key code. The default is 58_{16} for key F3 on an LK201 keyboard.
p3	A longword mask that specifies either a shift or control selection modifier key (INP$M_CONTROLMASK or INP$M_SHIFTMASK). The default selectors for the pseudomouse mode are CTRL/F3. If it is not used, **p3** must equal 0.
p4, p5, p6	Set to 0.

Set Screen Saver Timeout

The Set Screen Saver Timeout $QIO function enables or disables the screen saver function and sets the screen saver timeout value in seconds. The target device is the output monitor. Table 2–22 provides the required argument information for the Set Screen Saver $QIO call.

Table 2–22 Argument Data for Set Screen Saver $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_SCRSAV function modifier is used for setting a timeout value. An IO$M_DECW_RESET_SCRSAV optional function modifier is used to turn the screen saver off, or a IO$M_DECW_FORCE_SCRSAV function modifier is used to activate the screen saver.
p2	Specifies the timeout value in seconds. If **p2** is set to 0, the screen saver function is disabled.
p3, p4, p5, p6	Set to 0.

Ring Keyboard Bell

The Ring Keyboard Bell $QIO function rings the keyboard bell at a specified volume. The target device is the keyboard. Table 2–23 provides the argument information required for the Ring Keyboard Bell $QIO call.

Table 2–23 Argument Data for Ring Keyboard Bell $QIO Call

$QIO Argument	Required Data
func	IO$_SETMODE function code.
p1	IO$K_DECW_RING_BELL function modifier.
p2	Specifies the ring volume in percent. A value of 100 specifies the loudest ring while a 0 turns the bell off. A value of –1 provides a default volume of 70 percent or rings the bell at the current volume if previously set.
p3, p4, p5, p6	Set to 0.

3 Writing a Port Input Driver

Input driver software that makes up the device input DECwindows interface divides into two categories: class input drivers and port input drivers. This chapter describes the function, routines, and program entry points of a DECwindows port input driver.

More information concerning the data structures referenced in this chapter may be found in Appendix A. Macros are described in Appendix B. When you write a new driver according to the DECwindows requirements in this manual, consult the *VMS Device Support Manual* for basic driver design and the chapter on terminal class and port drivers for specific port/class information.

3.1 Overview

The port input driver is the device-dependent part of a DECwindows device driver. The port driver is sometimes referred to as the "interrupt" driver. It processes hardware interrupts and passes an uninterpreted byte stream of data to a class driver, where it is decoded into an X11 event packet.

The port driver contains device-dependent routines of a VMS terminal driver that are specific to a controller/CPU type. They bind together by means of the UCB to form a port/class interface for a DECwindows device-dependent driver.

The port driver contains the driver prologue table (DPT) data structure; initialization macros; device, unit, and controller initialization routines, a start I/O routine; port routines; and any additional device-dependent code, such as an interrupt service routine. VMS currently supports the port input driver (YEDRIVER) for workstations listed in Table 1–1.

The driver also contains the controller initialization and unit initialization routines that are startup routines required by VMS. They invoke macros needed by the port/class interface.

3.2 Port Driver Program Entry

Class drivers and output drivers call port routines to perform port-specific hardware functions. Routines of the port input driver are entered by way of a vector table. The port vector table is a data structure that allows the class driver to find the appropriate port routine. Each entry name is a specific vector table offset that points to the port routine. Therefore, each name is used as a symbolic offset. The port routine symbolic addresses in the table are as follows:

- PORT_STARTIO
- PORT_SET_LINE

- PORT_ABORT
- PORT_RESUME

The port driver builds the vector table by invoking the $VECINI macro, the $VEC macro for each table entry, and the $VECEND macro that terminates the structure. The COMMON_CTRL_INIT macro within the controller initialization routine relocates the table. Macros are described in Appendix B and the routines are in this chapter. A vectored port routine call example follows:

```
MOVL    UCB$L_TT_PORT(R5),R1 ;get vector table address
JSB     @PORT_STARTIO(R1)    ;call port start I/O routine
```

3.3 Port Input Driver Routines

The port input driver contains three types of routines: startup, initiate, and service routines. This section describes in alphabetical order the vectored routines that are part of the port input driver module.

PORT_ABORT

The PORT_ABORT routine commands the port to abort any currently active output activity. This port service routine may be called from the class input driver at any time and invalidates the data stored in UCB$L_TT_OUTADR.

input

Location	Contents
R5	Input UCB address

PORT_RESUME

The PORT_RESUME routine directs the port to resume any previously stopped output. The port must allow this routine to be called at any time (whether the output is active or was already stopped). This routine ensures that the hardware is enabled for output.

input

Location	Contents
R5	Input UCB address

PORT_SET_LINE

The PORT_SET_LINE routine changes the serial line characteristics. This initiate routine is called whenever any serial line characteristic in UCB$L_DEVDEPEND or UCB$L_DEVDEPEND2 is changed or when speed, parity, or automatic flow control are changed, or DMA is enabled/disabled. This is the only port routine that can write the fields UCB$L_DEVDEPEND and UCB$L_DEVDEPEND2.

input

Location	Contents
R5	Input UCB address
UCB$B_TT_MAINT	Maintenance parameters
UCB$B_TT_PARITY	Parity, stop bits, and frame size
UCB$B_TT_SPEED	Low byte that defines transmit speed, high byte that defines receive speed or is 0
UCB$B_TT_PRTCTL	DMA and AUTOXOFF-enable flags
UCB$L_DEVDEPEND	First longword for device-dependent status
UCB$L_DEVDEPND2	Second longword for device-dependent status

output

Location	Contents
R4	Destroyed

PORT_STARTIO

The PORT_STARTIO initiate routine starts output on a serial line that is currently inactive. It enables output interrupts on an idle controller unit. It is always called with a character key (data byte) or a burst of data.

input

Location	Contents
R3	Character to output (single character only)
R5	Input UCB address
UCB$B_TT_OUTYPE	0 if no character to output, 1 if one character to output, or a negative value if data burst to output
UCB$L_TT_OUTADR	Address of burst if UCB$B_TT_OUTYPE is negative
UCB$B_TT_OUTLEN	Length of data burst

output

Location	Contents
R0 through R4	Destroyed
R5	UCB address

Controller Initialization Routine

The controller initialization routine prepares a controller or hardware interface for operation. The routine is entered at system startup and during recovery after power failure and is always called at IPL$_POWER. The routine resets the controller unit. This routine invokes the COMMON_CTRL_INIT macro to relocate the driver vector table. The DPT_STORE macro places the address of this routine in the CRB.

Note that before invoking the COMMON_CTRL_INIT macro, a DECwindows port driver should invoke the $DECW_COMMON_READY macro to ensure that the common driver is loaded. If the common driver is not loaded, the driver does not operate on calls to the common service routines and the system may crash.

input

Location	Contents
R4	CSR address of the port
R5	IDB address of the controller unit
R6	DDB address of the controller unit
R8	CRB address of the controller unit

output

Location	Contents
R0, R1, R2	Destroyed

Unit Initialization Routine

The unit initialization routine sets up each individual device. The routine loads specific UCB locations with hardware unit requirements or controller-specific data, binds the class and port drivers, readies the hardware for input and output, and takes any necessary action should a power failure occur. The initialization routine loads the port vector table address into UCB field UCB$L_TT_PORT. This routine is invoked each time a unit is created. This routine is always called at IPL$_POWER. The DPT_STORE macro places the address of this routine in the CRB and sets the unit's DDT to the address of the common DDT.

input

Location	Contents
R4	CSR address of the unit
R5	Input UCB address of the unit

output

Location	Contents
R4	Preserved
R5	Preserved

4 Writing a Class Input Driver

Input driver software that makes up the device input DECwindows interface divides into two categories: class input drivers and port input drivers. This chapter describes the function, routines, and program entry of a DECwindows class input driver.

More information concerning the data structures referenced in this chapter may be found in Appendix A. Macros are described in Appendix B. When you write a new driver according to the DECwindows requirements in this manual, consult the *VMS Device Support Manual* for basic driver design and the chapter on terminal class and port drivers for specific port/class information.

4.1 Overview

The class input driver is the device-independent part of a device driver. A DECwindows class driver decodes serial device data and formats it into event packets for the server. The class driver is sometimes referred to as the "decoder" driver, as the serial data is decoded into events. Decoded events reported to the server include pointer motion, mouse button transitions, and key transitions.

A DECwindows class driver contains routines that implement the various device input, byte-stream, decoding functions that are independent of the controller/CPU type. These class routines specifically support the DECwindows standard interface. The port driver contains device-dependent routines of a VMS terminal driver that are specific to a controller/CPU type.

The DECwindows device drivers use only a subset of the full VMS terminal port/class interface. They bind together by means of the device UCB to form a port/class interface for a DECwindows device-dependent driver. DECwindows software currently supports the class input drivers for the keyboard and mouse (IKDRIVER, IMDRIVER) listed in Table 1–1.

The driver also contains controller initialization and unit initialization routines that are startup routines required by VMS. They invoke macros needed by the port/class interface.

4.2 Class Driver Program Entry

Routines of the class driver are entered by way of a vector table. The vector table is a data structure that allows other drivers to find the appropriate class routine. Each entry name is a specific offset that points to the class routine. Therefore, each name is used as a symbolic offset. The class routine symbolic addresses in the table are as follows:

- CLASS_PUTNXT
- CLASS_GETNXT
- CLASS_DDT

The class driver builds the vector table by invoking the $VECINI macro, the $VEC macro for each table entry, and the $VECEND macro that terminates the structure. The COMMON_CTRL_INIT macro within the controller initialization routine relocates the table. Macros are described in Appendix B and the routines are discussed in this chapter.

The class driver routines are entry points from the common, port, and output drivers. Driver calls to some of these routines for queue interface refer to symbolic offsets in the class vector table. A vectored class routine is called by a JSB instruction. Class routine call examples follow:

```
#1  MOVL  UCB$L_TT_CLASS(R5),R0 ;get vector table address
    JSB   @CLASS_PUTNXT(R0)     ;call class routine

#2  JSB   @UCB$L_TT_PUTNXT(R5)  ;use offset directly in UCB
                                ;to find routine
```

Note that, because the get-next-character and put-next-character routines are the most heavily used class driver routines, their addresses are stored in the terminal extension UCB. Fields UCB$L_TT_PUTNXT and UCB$L_TT_GETNXT provide direct access to the class driver. It is therefore possible to use one instruction (method 2), assuming R5 contains the UCB base address. This eliminates the move instruction (vector to general register) required in method 1.

4.3 Class Input Driver Routines

This section describes in alphabetical order the routines in a class input driver. The routines in this section are common to both the keyboard and mouse drivers (IKDRIVER and IMDRIVER).

CLASS_DDT

This entry in the class driver vector table points to the driver dispatch table (DDT). It is simply an offset to the table and not to a routine. The CLASS_UNIT_INIT macro uses the CLASS_DDT entry point to load the address of the DDT into the UCB. The DDT is described in the *VMS Device Support Manual*.

CLASS_GETNXT

The CLASS_GETNXT routine returns to the caller with the next byte to be output from the SILO buffer. The port driver calls CLASS_GETNXT whenever it has finished processing an output request to see if there are more output requests in the SILO buffer. The CLASS_GETNXT routines calls the GET_ONE_BYTE routine that gets the data byte in the SILO. For example, in a keyboard driver, this routine passes LED (light) information from the SILO buffer to the keyboard.

input

Location	Contents
R5	Input UCB address

output

Location	Contents
UCB$B_TT_OUTYPE	0 if there is no data to be output, 1 if one character is in R3, or a negative value if there is a data burst to output
UCB$L_TT_OUTADR	Address of burst if UCB$B_TT_OUTYPE is negative
UCB$B_TT_OUTLEN	Length of data burst
R5	UCB address
All other registers	Destroyed

CLASS_PUTNXT

The CLASS_PUTNXT routine is called by a port driver to pass input data from the serial line to the input queue. CLASS_PUTNXT decodes the data and converts it to an x event in the form of an input packet. It uses the GET_FREE_KB_PACKET macro to get a free packet and it terminates processing by invoking the PUT_INPUT_ON_QUEUE macro to insert the event in the input queue.

Note that the input and free queues are self-relative queues that must be accessed using interlocked instructions. The GET_FREE_KB_PACKET macro removes a free packet from the free queue using the REMQHI instruction. The routine then decodes the byte stream setting the event data in the packet, and the PUT_INPUT_ON_QUEUE macro inserts the packet in the input queue using the INSQTI instruction.

input

Location	Contents
R0	CSR
R3	Input data byte
R5	Input UCB address

output

Location	Contents
UCB$B_TT_OUTYPE	0 if no data to be output, 1 if one character in R3, or a negative value if data burst to output
UCB$L_TT_OUTADR	Address of first character in burst (burst mode only)
UCB$W_TT_OUTLEN	Length of data burst (burst mode only)
R0	Preserved
R5	UCB address
All other registers	Destroyed

Controller Initialization Routine

The controller initialization routine prepares a controller or hardware interface for operation. The routine is entered at system startup and during recovery after power failure and is always called at IPL$_POWER. The routine resets the controller unit. This routine invokes the COMMON_CTRL_INIT macro to relocate the driver vector table.

Note that before invoking the COMMON_CTRL_INIT macro, a DECwindows class driver should invoke the $DECW_COMMON_READY macro to ensure that the common driver is loaded. If the common driver is not loaded, the driver does not operate on calls to the common service routines and the system may crash.

input

Location	Contents
R4	CSR address of the controller
R5	IDB address of the controller
R6	DDB address of the controller
R8	CRB address of the controller

output

Location	Contents
R0, R1, R2	Destroyed

Unit Initialization Routine

The unit initialization routine in the class driver sets up each device unit. The routine loads specific UCB locations with hardware unit requirements or controller-specific data, binds the class and port drivers, readies the hardware for input and output, and takes any necessary action should a power failure occur. The initialization routine loads the class vector table address into UCB field UCB$L_TT_CLASS. The unit initialization routine calls the COMMON_UNIT_INIT macro, which sets the driver dispatch table address (CLASS_DDT) in the class vector table to that of the common DDT. This routine is run each time a unit is created. This routine is always called at IPL$_POWER.

input

Location	Contents
R4	CSR address of the unit
R5	Input UCB address of the unit

output

Location	Contents
R4	Preserved
R5	Preserved

5 Common Driver

This chapter describes the function, routines, and program entry of the DECwindows common driver. The chapter also describes the routines that service the $QIO interface. The $QIO calls are described in Chapter 2 and Chapter 6. More information concerning the data structures referenced in this chapter may be found in Appendix A.

5.1 Overview

The DECwindows common device driver (INDRIVER) is the DECwindows server interface to the input and output drivers and performs the device-independent functions of a workstation. The common device interface comprises routines, an input queue for the primary server interface, and a $QIO interface.

The common driver contains a set of DECwindows routines called by output drivers and class input drivers. It also contains routines required by VMS for all device drivers to communicate with the VMS system. Function decision table (FDT) routines within the driver service the $QIO interface.

DECwindows output and class drivers locate and call common driver service routines directly, through the common vector table, or indirectly, by invoking macros. For example, moving the mouse may necessitate moving the cursor. Because cursor movement is controlled by the graphics controller, class drivers access this function in the output driver by way of the common driver.

The common driver services and macros provided are listed in Table 5–1. As outlined in the table, the common driver routine descriptions that follow in this chapter are grouped according to type of service.

Table 5–1 Common Driver Services

Type of Service	Symbol	Description
Class	COMMON_CTRL_INIT	Macro to relocate vector table
Class	COMMON_UNIT_INIT	Macro to set DDT to class DDT
Class	COMMON_POS_CURSOR	Routine to position cursor
Class	COMMON_SETUP_INPUT_UCB	Routine to set up input UCB
Output	COMMON_CTRL_INIT	Macro to relocate vector table
Output	COMMON_UNIT_INIT	Macro to set DDT to class DDT
Output	COMMON_SETUP_OUTPUT_UCB	Routine to set up output UCB

(continued on next page)

Table 5–1 (Cont.) Common Driver Services

Type of Service	Symbol	Description
Output	COMMON_VSYNC	Routine to perform VSYNC timed functions
VMS	CONTROL_INIT	Controller initialization routine
VMS	UNIT_INIT	Unit initialization routine
VMS	CANCEL	Cancel routine
VMS data	COMMON_DDT	Driver dispatch table
VMS data	COMMON_FLAGS	Common flag-bits data word
Type of Service	**Mode**	**Description**
VMS FDT	Output request	Routine for buffered I/O output preprocessing
VMS FDT	Output request	Routine for direct I/O output preprocessing
$QIO	Sense	Routine for sense mode of device
$QIO	Set	Routine for set mode of device

5.2 Common Driver Program Entry

DECwindows routines in the common driver for class and output services are entered by means of a vector table. The common vector table is a data structure that allows a driver to find the appropriate service routine. Each entry name is a specific vector table offset (relative to the beginning of the common driver) that points to the common routine. Therefore, each name may be used as a symbolic offset into the table. The service routine symbolic addresses appear in the table as follows:

- COMMON_DDT
- COMMON_POS_CURSOR
- COMMON_SETUP_INPUT_UCB
- COMMON_SETUP_OUTPUT_UCB
- COMMON_VSYNC
- COMMON_FLAGS

The common driver builds the vector table by invoking the $VECINI macro, the $VEC macro for each table entry, and the $VECEND macro that terminates the structure. A macro within the controller initialization routine relocates the table. The routines are described in this chapter and the macros in Appendix B.

Driver calls to these common vectored routines refer to symbolic offsets in the common vector table. When INDRIVER is loaded, DECW$GL_VECTOR is a global location that contains the address of the common vector table. A common routine call example follows:

```
MOVL  G^DECW$GL_VECTOR,R0 ;get global pointer to vector table
JSB   @COMMON_VSYNC(R0)   ;call the COMMON_VSYNC routine
```

5.3 Common Driver Routines for Class Service

The common driver module contains class service routines that are required by class drivers. This section presents the common driver class service routines in alphabetical order. The class service routines are as follows:

- COMMON_POS_CURSOR
- COMMON_SETUP_INPUT_UCB

COMMON_POS_CURSOR

The COMMON_POS_CURSOR routine positions the cursor (pointer) according to the location specified in the UCB (UCB$W_DECW_CURSOR_X and UCB$W_DECW_CURSOR_Y). The cursor position adjusts to the hotspot and is clipped to the physical screen boundaries.

input

Location	Contents
R5	Output UCB address

COMMON_SETUP_INPUT_UCB

The COMMON_SETUP_INPUT_UCB routine clears most fields of the class input UCB and sets the system defaults where required. It is called by the port driver's unit initialization routine.

input

Location	Contents
R0	A value that determines which characteristics to set as follows: **Value** — **Meaning** 0 — An unknown device +1 — System keyboard –1 — System pointing device
R5	Address of the input UCB being initialized

output

Location	Contents
R0, R1, R2	Destroyed

5.4 Common Driver Routines for Output Service

The common driver module contains output service routines that are common among output drivers. This section presents the common driver output service routines in alphabetical order. The output service routines are as follows:

- COMMON_SETUP_OUTPUT_UCB
- COMMON_VSYNC

COMMON_SETUP_OUTPUT_UCB

The COMMON_SETUP_OUTPUT_UCB routine clears most fields of the output UCB and sets the system defaults where required (screen saver and starting cursor position). This routine is called by an output driver's unit initialization routine. COMMON_SETUP_OUTPUT_UCB allocates and initializes a device information block (DVI) and then calls the OUTPUT_SET_DVI routine, which loads device-specific values into the DVI fields.

input

Location	Contents
R5	Output UCB address

output

Location	Contents
R0	Status

COMMON_VSYNC

The COMMON_VSYNC routine performs time-sensitive processing during the vertical synchronization (VSYNC) interval. Because most output drivers receive a vertical SYNC interrupt to perform the necessary hardware functions, the output driver also calls the COMMON_VSYNC routine from its interrupt service routine. Because this call is inherently timer based, the COMMON_VSYNC routine periodically checks whether the screen saver function should be enabled or disabled. It also checks the input queue at this interval and determines whether the server needs to be notified of new input.

input

Location	Contents
R5	Output UCB address

output

Location	Contents
R0	Status
All other registers	Preserved

5.5 Common Driver Vectored Data

The common driver module contains vectored data structures that are required for DECwindows global driver service. This section presents the vectored data segments. Offset entry names to vectored data segments are as follows:

- COMMON_DDT
- COMMON_FLAGS

COMMON_DDT

This entry in the common driver vector table points to the driver dispatch table (DDT). It is simply an offset to the DDT and not a routine. The COMMON_UNIT_INIT macro uses the COMMON_DDT entry point when loading the address of the DDT into the UCB. The DDT is described in the *VMS Device Support Manual*.

COMMON_FLAGS

This entry in the common driver vector table points to the driver common flags longword (COMMON_FLAG_BITS). It is simply an offset to the common flag bits or global flags and not a routine. The $DECW_COMMON_READY macro uses the flag bits to sense whether the drivers are loaded. The common bits in the flags longword are listed in Table 5–2.

Table 5–2 Common Flags Word

Bits	Description
FLAGS$V_KB_DECODER	Bit 0 is set when the keyboard class driver is loaded.
FLAGS$V_PTR_DECODER	Bit 1 is set when the pointer class driver is loaded.

5.6 Common Driver FDT Routines for $QIO Service

The $QIO common interface provides for initialization and information requests from a server or extension to a device. The common driver module contains function decision table (FDT) routines that service the $QIO calls in the common interface. The specific requirements and form of each $QIO call are described in Chapter 2. This section discusses the functions of the common driver FDT routines.

5.6.1 General

When a user process calls the SYS$QIO system service, the system service uses the I/O function code specified in the request to scan the driver FDT and selects one or more of the FDT routines provided by the common driver. The common driver performs device-independent processing and then calls the output or input driver for any device-dependent processing. For example, during a cursor pattern $QIO service, the common driver validates the parameters and then calls the appropriate output driver to change the cursor.

To prepare for an I/O operation, FDT routines perform such tasks as allocating buffers in system space and validating the device-dependent arguments (**pl** through **p6**) of the I/O request.

Before calling an FDT routine, the $QIO system service sets up the contents of certain registers, as listed in Table 5–3.

Table 5–3 Registers Loaded by the $QIO System Service

Register	Contents
R0	Address of FDT routine being called
R3	Address of IRP for current I/O request
R4	Address of process control block (PCB) of current process
R5	Address of UCB of device assigned to user-specified process-I/O channel
R6	Address of CCB that describes user-specified process-I/O channel
R7	Bit number of user-specified I/O function code
AP	Address of first $QIO parameter (**p1**)

The common driver contains the preprocessor FDT routines and uses the FUNCTAB macro to build a function decision table that lists valid function codes for a given preprocessor. These function codes are entry points to the FDT routines that perform I/O processing for each function specified in the $QIO service call. A list of the common driver FDT function codes and functions is shown in Table 5–4.

Table 5–4 Common Driver FDTs and Function Codes

Function Code	Description
IO$_SENSEMODE	Sense mode of device
IO$_SENSECHAR	Sense device characteristics
IO$_SETMODE	Set mode of device
IO$_SETCHAR	Set device characterisitics
IO$_DECW_OUTPUT_BUFFERED_FDT	Preprocess buffered I/O output FDT
IO$_DECW_OUTPUT_DIRECT_FDT	Preprocess direct I/O output FDT

The following sections describe the $QIO functions and modifiers processed by the device-dependent FDT routines that make up the $QIO common interface.

5.6.2 FDT Sense-Mode Routines

The sense-mode FDT routines service $QIO requests to retrieve device information or characteristics. The function modifier passed in the **pl** argument selects the appropriate subroutine in the FDT sense-mode routine table for the specific sense-mode function. When the FDT routine exits, it either queues the I/O request, finishes processing the I/O, or aborts the I/O. The following are valid function services as listed in the FDT_SENSEM table that point to the appropriate sense-mode subroutine.

- DEVICE_INFO
- SENSE_KB_INFO
- SENSE_KB_LED
- SENSE_PMOUSE_KEY
- SENSE_PTR_ACCEL
- SENSE_OPWIN_KEY
- SENSE_MOTION_COMP
- SENSE_SCREEN_SAVER

5.6.3 FDT Set-Mode Routines

The set-mode FDT routines service $QIO requests to set various device characteristics. The function modifier passed in the **pl** parameter selects the appropriate subroutine in the FDT set-mode routine table for the specific set-mode function. When the FDT routine exits, it either queues the I/O request, calls the output driver to complete the request, finishes processing the I/O, or aborts the I/O. The following are valid function services as listed in the FDT_SETM table that point to the appropriate set-mode subroutine.

- ENABLE_INPUT
- MOTION_BUFFER_INIT

- SET_ATTACH_SCREEN
- SET_CURSOR_PATTERN
- SET_CURSOR_POS
- SET_KB_INFO
- SET_KB_LED
- SET_MOTION_COMP
- SET_OPWIN_KEY
- SET_PMOUSE_KEY
- SET_PTR_ACCEL
- SET_SCREEN_SAVER
- RING_BELL

5.6.4 FDT Output Routines

The FDT common driver output routines provide $QIO services that preprocess a $QIO output request. The output routines check whether the device being addressed is capable of providing output, then vectors the request to the appropriate FDT parsing routine in the output driver (using the output vector table). The routines preprocess both direct I/O and buffered I/O output requests.

- FDT_OUTPUT_B
- FDT_OUTPUT_D

6 Output Driver

This chapter describes the vectored output routines that process graphics requests from the server to the screen. The output driver operates on the video controller that manages output functions of the graphics hardware. Details for the standard VMS routines may be found in the VMS documentation. Macros called by the output routines are described in Appendix B. More information concerning data structures referenced in this chapter may be found in Appendix A.

6.1 Overview

The current DECwindows output device drivers (GxxDRIVER modules) provide device-dependent driver support for monochrome and color graphics (GPX) controllers. The drivers and their associated workstations are shown in Table 1–1. The drivers provide services for DECwindows and VMS, as well as services for $QIO requests for output to screen displays. The output driver for a color (GPX) workstation interprets direct memory access (DMA) packets from the server and presents the packet data to the graphics hardware. The output driver for monochrome workstations uses a monochrome frame buffer (MFB) for issuing output data to the graphics hardware. The drivers may contain routines, macros, and services to execute draw requests, copy data between host memory and video memory, load template RAM, manipulate the cursor, modify the color map, and get or set device-specific information for $QIO requests.

6.2 Output Driver Program Entry

DECwindows routines of the output driver are entered by way of a vector table. The output vector table is a data structure that allows the common driver to find the appropriate service routine. Each entry name is an address (relative to the beginning of the output driver prologue table) of a service routine. Each name may be used as a symbolic offset into the table. The service routine symbolic addresses in the table appear as follows:

- OUTPUT_CLEAR_CURSOR
- OUTPUT_CURSOR_PATTERN
- OUTPUT_DISABLE_VIDEO
- OUTPUT_ENABLE_VIDEO
- OUTPUT_BUFFERED_FDT
- OUTPUT_POS_CURSOR
- OUTPUT_CANCEL
- OUTPUT_DIRECT_FDT

- OUTPUT_SET_DVI
- OUTPUT_OPWIN_VISIBLE
- OUTPUT_OPWIN_UP
- OUTPUT_OPWIN_DOWN
- OUTPUT_OPWIN_RESIZE

The output driver builds the vector table by invoking the $VECINI macro, the $VEC macro for each table entry, and the $VECEND macro that terminates the structure. The macro COMMON_CTRL_INIT within the controller initialization routine relocates the table. Macros are described in Appendix B and the routines are described in this chapter. A vectored output routine call example follows:

```
MOVL  UCB$L_DECW_OUTPUT_VECTOR(R5),R1 ;get vector table address
JSB   @OUTPUT_CURSOR_PATTERN(R1)      ;load the cursor pattern
```

6.3 Output Driver Routines

The vectored output routines provide video and cursor image control, operator window control, start and cancel I/O, and device-dependent $QIO service. Table 6–1 lists and briefly describes the vectored routines that are part of the code of the output driver module. When called, all routines assume that R5 contains the output UCB starting address. The OPWIN (operator window) routines in the table are only valid for workstations that are the active system console.

Table 6–1 Output Vector Table Routines

Vector Name	Routine Function
OUTPUT_CLEAR_CURSOR	The clear-cursor routine redraws the cursor with all zeros (nulls the cursor). (Not used in the GPX drivers.)
OUTPUT_CURSOR_PATTERN	The cursor-pattern routine sets or loads the cursor pattern in UCB$L_DECW_CURSOR_PATTERN of the UCB output extension.
OUTPUT_DISABLE_VIDEO	The disable-video routine and macro disables the video display output (for example, screen-save).
OUTPUT_ENABLE_VIDEO	The routine and macro enable the video display output.
OUTPUT_BUFFERED_FDT	The buffered-FDT routine parses $QIO calls from the common driver that specify the buffered output mode of $QIO processing in the function code. The routine also checks the validity of the function code in the **p1** parameter being passed in the call.

(continued on next page)

Table 6–1 (Cont.) Output Vector Table Routines

Vector Name	Routine Function
OUTPUT_POS_CURSOR	The routine positions the cursor on the screen according to the *x*- and *y*-coordinates defined in fields UCB$W_QD_CURSOR_X and UCB$W_QD_CURSOR_Y.
OUTPUT_CANCEL	The FDT cancel routine is called by the common driver to cancel all IRPs in the wait queue. The routine cancels or aborts all outstanding I/O requests.
OUTPUT_DIRECT_FDT	The direct-FDT routine parses $QIO calls from the common driver that specify the direct output mode of $QIO processing. The routine also validates the function code in the **p1** parameter being passed in the call.
OUTPUT_SET_DVI	The routine sets or initializes fields in the device information block (DVI).
OUTPUT_OPWIN_VISIBLE	Vector to the check-for-operator-window-mode-capability routine in the output driver subroutine module.
OUTPUT_OPWIN_UP	Vector to the display-operator-window routine in the output driver subroutine module.
OUTPUT_OPWIN_DOWN	Vector to the remove-operator-window routine in the output driver subroutine module.
OUTPUT_OPWIN_RESIZE	Vector to the resize-operator-window routine in the output driver subroutine module. Makes the window smaller to adapt to the window system.

6.4 Queue Processing of Output

The mechanism for passing data between server and output driver is an output queue. The output queue is a pair of interlocked queues: a GPX packet buffer (GPB) command queue and a GPB free queue. The queues consist of GPB packet buffers and their associated control blocks. The GPB buffers contain various command packets generated for the server output. These structures exist in nonpaged memory and are accessed by the server by mapping PFNs into P0 process space.

Like the common driver and input queue, the server and output driver use an interlocked queue instruction to remove data packet buffers from the command queue (REMQHI) and to insert free packets on the free queue (INSQTI).

6.5 $QIO Output Interface

The GADRIVER module contains function decision table (FDT) routines that provide a $QIO output interface. The $QIO interface is for color/DMA drivers only. A main FDT parsing routine and its FDT subroutines service all incoming $QIO calls from the server/common driver interface. The output driver does not maintain its separate FDT table. The specific FDT routines are accessed through the vectored routines that initialize and manipulate the output packet data structures.

6.6 $QIO Calls to Output Driver

This section presents the output $QIO calls used in a server that are supported by services within the DECwindows output driver. The $QIO system service format is presented first.

$QIO calls must be issued to a physical device. Initially, using the $ASSIGN system service, the appropriate device name is assigned to an I/O channel. The channel number entry is required for **chan** in the $QIO system service call. Physical device names on a GPX workstation are GAA0 for output to the screen, GAA1 for the keyboard, and GAA2 for the mouse (see Table 1–1). Note that the DECwindows environment provides logical names (DECW$WORKSTATION, DECW$KEYBOARD, and DECW$POINTER) to point to the physical device.

$QIO System Service

The Queue I/O Request service queues an I/O request to a channel associated with a device. The SYS$QIO format described next applies to all the $QIO calls presented in this chapter. For more information on SYS$QIO refer to *VMS System Services Reference Manual.*

FORMAT

SYS$QIO *[efn],chan,func,[iosb],[astadr],[astprm] ,p1,p2,p3[,p4][,p5][,p6]*

arguments

efn is the event flag number of the I/O operation. The **efn** argument is a longword containing the number of the event flag.

chan is the I/O channel assigned ($ASSIGN) to the device name to which the request is directed. The **chan** argument is a longword containing the number of the I/O channel; however, $QIO uses only the low-order word.

func is the device-specific function code specifying the operation to be performed. The **func** argument is a longword containing the function code.

iosb is the I/O status block to receive the final completion status of the I/O operation. The **iosb** argument is the address of the quadword I/O status block.

astadr is the AST service routine to be executed when the I/O completes. The **astadr** argument is the address of a longword that is the entry mask to the AST routine.

astprm is the AST parameter to be passed to the AST service routine. The **astprm** argument is a longword containing the AST parameter.

pl is the function modifier specifying the specific service being called within the basic function code.

p2 to **p6** are the function-specific parameters being passed.

6.7 Output $QIO Calls

The $QIO interface is for color/DMA drivers only. Using $QIO calls, DMA packet data passes directly to the output driver using a set of packet data structures in an output queue that is created by the output driver. The following $QIO calls that create the output queue interface are supported by the DECwindows output driver.

- Create GPD
- Queue GPB
- GPB Wait

This section defines the specific argument information required for the various $QIO calls serviced by the output driver. The calls use the IO$_DECW_OUTPUT_DIRECT_FDT or the IO$_DECW_OUTPUT_BUFFERED_FDT function code.

Create GPD

The create GPX physical data (Create GPD) $QIO function creates a GPX shared memory data block. A $QIO call to this service allocates and maps a contiguous region of memory from nonpaged pool for the entire output queue holding all GPBs. The region is mapped so it is accessible by the hardware and VMS. The block size in pages must be specified in **p2**. If a GPD already exists, the GPD block is reinitialized. Table 6–2 provides the required argument information for the Create GPD $QIO call. The service returns **p2** and **p3** parameters defining the existing block size and starting PFN, and the service also updates the I/O status block (IOSB) with the same information.

Table 6–2 Argument Data for Create GPD $QIO Call

$QIO Argument	Required Data
func	IO$_DECW_OUTPUT_DIRECT_FDT or IO$_DECW_OUTPUT_BUFFERED_FDT function code.
p1	IO$K_DECW_CREATE_GPD function modifier.
p2	Size of the GPX physical data block in pages.
p3	Page frame number (PFN) of the first page of the data block. (A return parameter only. Within the first page is a table listing the remaining PFNs that make up the GPD.)

Queue GPB

The queue GPX packet buffer (Queue GPB) $QIO function inserts a newly formed GPB in the command queue. This service is also known as INSQTI GPB, because of its insert-on-queue function. Table 6–3 provides the required argument information for the Queue GPB $QIO call.

Table 6–3 Argument Data for Queue GPB $QIO Call

$QIO Argument	Required Data
func	IO$_DECW_OUTPUT_DIRECT_FDT or IO$_DECW_OUTPUT_BUFFERED_FDT function code
p1	IO$K_DECW_INSQTI_GPB function modifier
p2	Offset of the new GPB within the GPD data block
p3	Start of user buffer (optional)
p4	Length of user buffer (optional)

GPB Wait

The GPX packet buffer wait (GPB Wait) $QIO function suspends the I/O request when there are no free GPX packet buffers. The $QIO completes when there are one or more GPBs on the free queue. Table 6–4 provides the required argument information for the GPB Wait $QIO call.

Table 6–4 Argument Data for GPB Wait $QIO Call

$QIO Argument	Required Data
func	IO$_DECW_OUTPUT_DIRECT_FDT or IO$_DECW_OUTPUT_BUFFERED_FDT function code
p1	IO$K_DECW_GPBWAIT function modifier

A Data Structures

The DECwindows device driver software requires an I/O database that contains both the standard VMS data structures and data structures specific to DECwindows. The data structures provide information to the VMS operating system and to drivers that help monitor status of and control the functions of the I/O subsystem. All of the data structures are allocated space in nonpaged system memory. DECwindows common and input driver structures are defined by the DECwindows global macro $DECWGBL. The data structures in the following list are described in this appendix. These are the specific sources of data for the DECwindows device drivers.

DVI	Device information block
INB	Input buffer control block
INP	Input packet structure
KIB	Keyboard information block
MHB	Motion history buffer
UCB/DWI/DECW	Unit control block, DECwindows common input extension
UCB/DWI/KB	Unit control block, DECwindows keyboard input extension
UCB/DWI/PTR	Unit control block, DECwindows pointer input extension
UCB/DECW	Unit control block, DECwindows common output extension
Vector Tables	For class, common, port, and output drivers

The VMS data structures listed below are also used by DECwindows device driver modules. These data structures are described in the *VMS Device Support Manual* and the *VAX/VMS Internals and Data Structures* book.

ACB	AST control block
CRB	Channel request block
DDB	Device data block
DDT	Driver dispatch table
DPT	Driver prologue table
FDT	Function decision table
IDB	Interrupt dispatch block
IRP	I/O request packet
ORB	Object rights block
UCB	Unit control block (main system portion)
UCBx	Unit control block (class/port terminal extensions)

A.1 Device Information Block (DVI)

The device information block (DVI) in the I/O database provides DECwindows device-specific information, primarily concerning the color graphics (GPX) workstation. The fields define system defaults and static device information and are read access only (see Figure A–1). Table A–1 lists and defines the fields of the block. The length of the data structure is defined by the constant DVI$K_LENGTH.

Figure A–1 Device Information Block (DVI)

DVI$L_FLINK				0
DVI$L_BLINK				4
DVI$B_SUB_TYPE	DVI$B_TYPE	DVI$W_SIZE		8
DVI$L_FLAGS				12
DVI$B_SPARE_1	VSYNC_INTERVAL	COMPRESSION_TYPE	DVI$B_WS_TYPE	16
DVI$L_SCRSAV_TIMEOUT				20
DVI$W_COLOR_MAP_ENTRIES		DVI$B_COLOR_MAPS	SYSTEM_COLOR_MAP	24
DVI$B_CI_SPARE_1	DVI$B_CURSORS	SYSTEM_CURSOR	DVI$B_BITS_PER_RGB	28
DVI$B_CURSOR_PLANES (16 bytes)				32
DVI$B_CURSOR_WIDTH (16 bytes)				48
DVI$B_CURSOR_HEIGHT (16 bytes)				64
DVI$B_CURSOR_TYPE (16 bytes)				80
DVI$L_VIDEO_STARTING_PFN				96
BITS_PER_PIX_BTP	BITS_PER_PIXEL	DVI$W_VIDEO_PAGE_COUNT		100
DVI$W_VIDEO_MEM_HEIGHT		DVI$W_VIDEO_MEM_WIDTH		104
DVI$W_ONSCREEN_Y_ORIGIN		DVI$W_ONSCREEN_X_ORIGIN		108
DVI$W_ONSCREEN_HEIGHT		DVI$W_ONSCREEN_WIDTH		112

(continued on next page)

Figure A–1 (Cont.) Device Information Block (DVI)

DVI$W_OPWIN_Y_ORIGIN	DVI$W_OPWIN_X_ORIGIN	116
DVI$W_OPWIN_HEIGHT	DVI$W_OPWIN_WIDTH	120
DVI$W_TABLET_HEIGHT	DVI$W_TABLET_WIDTH	124
DVI$L_LEGSS_START		128
DVI$L_LEGSS_SIZE		132
DVI$L_LEGO_P0_MASK		136
	DVI$W_LEGO_P0	

Table A–1 Device Information Block Fields

Field Name	Contents
DVI$L_FLINK	The forward link to the next DVI structure (not implemented).
DVI$L_BLINK	The backward link to the previous DVI structure (not implemented).
DVI$W_SIZE	Total size in bytes of the device information block.
DVI$B_TYPE	Defines the system or major type of data structure (DECwindows) that is read by the System Dump Analyzer (SDA). The common driver writes the symbolic constant DYN$C_DECW in this field when the common driver creates the DVI.
DVI$B_SUB_TYPE	Defines the specific (subtype) data structure (DVI) within the major type that is read by the SDA. The common driver writes the symbolic constant DYN$C_DECW_DVI in this field when the common driver creates the DVI.
DVI$L_FLAGS	A 32-bit field containing screen status bits. Mask bits in this field correspond to two possible states: DVI$V_PSEUDO_COLOR — Bit 0, set if any color maps exist. DVI$V_SCRSAV_ENABLED — Bit 1, set if screen saver is enabled.
DVI$B_WS_TYPE	The code for the workstation type.
DVI$B_COMPRESSION_TYPE	An 8-bit field containing status bits concerning device compression characteristics. Mask bits in this field correspond to two possible states: DVI$V_NONE — Bit 0, set if device hardware does not perform bit compression. DVI$V_FCC — Bit 1, set if device hardware has the FCC bit-compression feature.
DVI$B_VSYNC_INTERVAL	The vertical synchronization (VSYNC) interval in milliseconds.
DVI$L_SRCSAV_TIMEOUT	The screen saver timeout in seconds.
DVI$B_SYSTEM_COLOR_MAPS	The default number of the system color map.

(continued on next page)

Table A–1 (Cont.) Device Information Block Fields

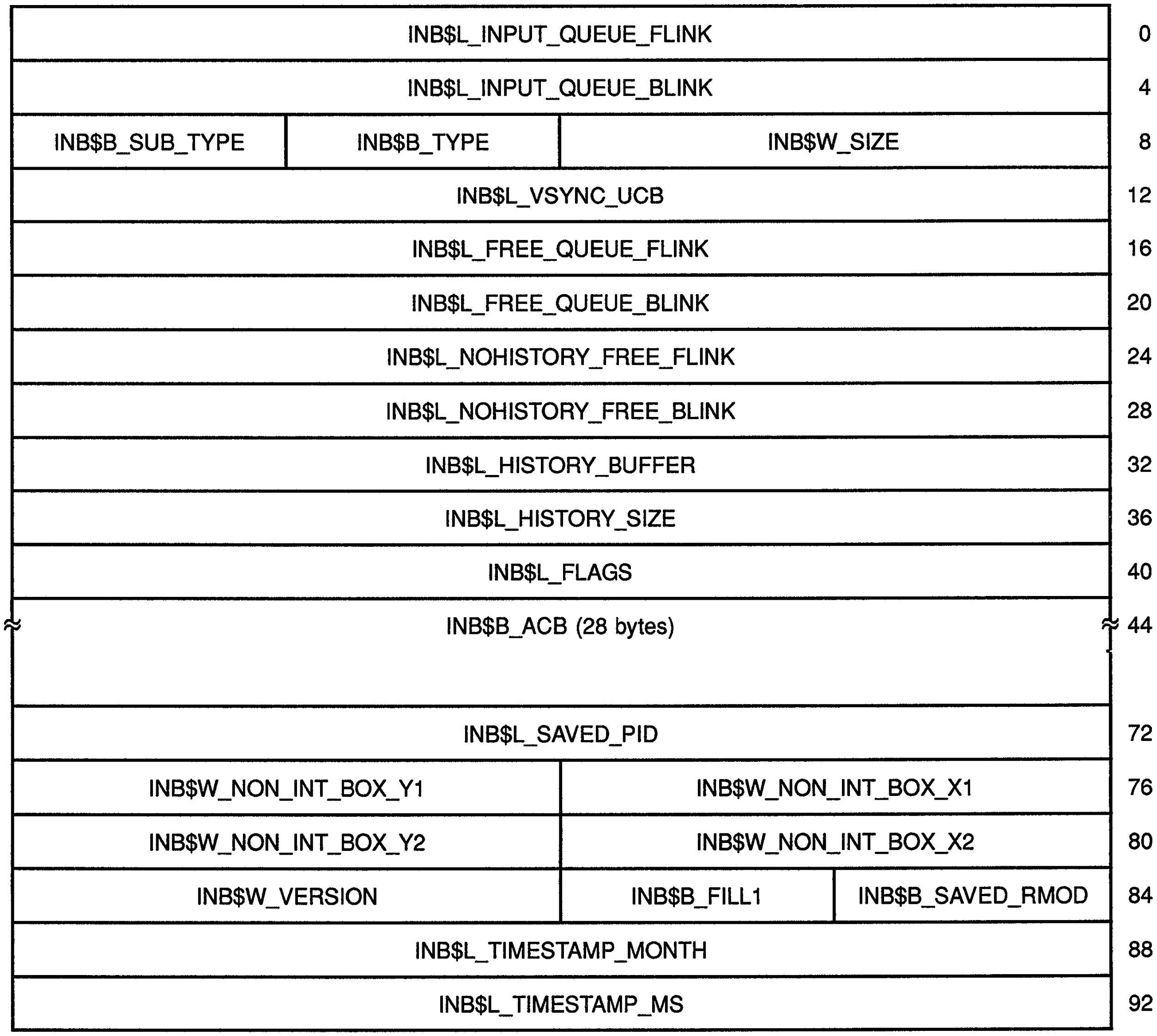
Field Name	Contents
DVI$B_COLOR_MAPS	The number of color maps.
DVI$W_COLOR_MAP_ENTRIES	The maximum number of colormap entries.
DVI$B_BITS_PER_RGB	The number of bits for every red/green/blue (RGB) pixel.
DVI$B_SYSTEM_CURSOR	The default system cursor number.
DVI$B_CURSORS	The number of hardware cursors.
DVI$B_CURSOR_PLANES	The address of the array (MAX16_CURSOR) for planes in the hardware cursor. (The last 15 bytes are reserved.)
DVI$B_CURSOR_WIDTH	The address of the array (MAX16_CURSOR) for the width of the cursor pattern. (The last 15 bytes are reserved.)
DVI$B_CURSOR_HEIGHT	The address of the array (MAX16_CURSOR) for the height of the cursor pattern. (The last 15 bytes are reserved.)
DVI$B_CURSOR_TYPE	The address of the array (MAX16_CURSOR) for the cursor type in the hardware cursor. Contains the DVI$V_COLOR_CURSOR bit, which is set if the cursor has color or is clear if the cursor is monochrome.
DVI$L_VIDEO_STARTING_PFN	A signed longword specifying the starting page frame number.
DVI$W_VIDEO_PAGE_COUNT	A signed word indicating the number of pages in the monochrome frame buffer.
DVI$B_BITS_PER_PIXEL	The number of bits for every pixel in the monochrome frame buffer.
DVI$B_BITS_PER_PIX_BTP	The number of bits for every pixel in the bitmap-to-processor (BTP) operation.
DVI$W_VIDEO_MEM_WIDTH	Screen width in pixels.
DVI$W_VIDEO_MEM_HEIGHT	Screen height in pixels.
DVI$W_ONSCREEN_X_ORIGIN	Defines the left screen margin (offset) in pixels to the visible screen point (visible 0).
DVI$W_ONSCREEN_Y_ORIGIN	Defines the top screen margin (offset) in pixels to the visible screen point (visible 0).
DVI$W_ONSCREEN_WIDTH	Defines the width in pixels of the visible screen (frame buffer width).
DVI$W_ONSCREEN_HEIGHT	Defines the height in pixels of the visible screen (frame buffer height).
DVI$W_OPWIN_X_ORIGIN	Defines the left screen margin (offset) in pixels to the start of the operator window.
DVI$W_OPWIN_Y_ORIGIN	Defines the top screen margin (offset) in pixels to the start of the operator window.
DVI$W_OPWIN_WIDTH	Defines the width in pixels of the operator window.
DVI$W_OPWIN_HEIGHT	Defines the height in pixels of the operator window.
DVI$W_TABLET_WIDTH	Defines the width of the tablet.
DVI$W_TABLET_HEIGHT	Defines the height of the tablet.
DVI$L_LEGSS_START	Reserved.
DVI$L_LEGSS_SIZE	Reserved.
DVI$L_LEGO_P0_MASK	Reserved.
DVI$W_LEGO_P0	Reserved.

A.2 Input Buffer Control Block (INB)

The input buffer control block (INB) in the I/O database defines the control block for the input queue buffer. INB provides control of the server/driver interface and pointers to the input queue and free packets (see Figure A–2). Table A–2 lists and defines the fields of the buffer.

One ACB (AST control block) is required for each input queue. An ACB is allocated at offset INB$B_ACB within the INB structure. It is defined by $ACBDEF and the 28-byte length is specified by ACB$K_LENGTH. The INB is defined in module $DECWCOMMON and the length is specified by INB$K_LENGTH.

Figure A–2 Input Buffer Control Block (INB)



INB$L_INPUT_QUEUE_FLINK				0
INB$L_INPUT_QUEUE_BLINK				4
INB$B_SUB_TYPE	INB$B_TYPE	INB$W_SIZE		8
INB$L_VSYNC_UCB				12
INB$L_FREE_QUEUE_FLINK				16
INB$L_FREE_QUEUE_BLINK				20
INB$L_NOHISTORY_FREE_FLINK				24
INB$L_NOHISTORY_FREE_BLINK				28
INB$L_HISTORY_BUFFER				32
INB$L_HISTORY_SIZE				36
INB$L_FLAGS				40
INB$B_ACB (28 bytes)				44
INB$L_SAVED_PID				72
INB$W_NON_INT_BOX_Y1		INB$W_NON_INT_BOX_X1		76
INB$W_NON_INT_BOX_Y2		INB$W_NON_INT_BOX_X2		80
INB$W_VERSION		INB$B_FILL1	INB$B_SAVED_RMOD	84
INB$L_TIMESTAMP_MONTH				88
INB$L_TIMESTAMP_MS				92

(continued on next page)

Figure A–2 (Cont.) Input Buffer Control Block (INB)

INB$W_SCHED_FLAGS	INB$W_SCHED_QUANTUM	96
INB$L_COUNTER1		100
INB$L_COUNTER2		104
INB$L_COUNTER3		108
INB$L_COUNTER4		112
INB$L_PFN_COUNT		116
INB$L_PFN_LIST		120

Table A–2 Input Buffer Control Block Fields

Field Name	Contents
INB$L_INPUT_QUEUE_FLINK	The forward link to the first or next input packet structure (INP) in the input queue.
INB$L_INPUT_QUEUE_BLINK	The backward link to the last or previous input packet structure (INP) in the input queue.
INB$W_SIZE	Total size in bytes of the input buffer.
INB$B_TYPE	Defines the system or major type of data structure (DECwindows) that is read by the System Dump Analyzer (SDA). The common driver writes the symbolic constant DYN$C_DECW in this field when the common driver creates the INB.
INB$B_SUB_TYPE	Defines the specific (subtype) data structure (INB) within the major type that is read by the SDA. The common driver writes the symbolic constant DYN$C_DECW_INB in this field when the common driver creates the INB.
INB$L_VSYNC_UCB	The pointer to the output UCB whose driver first reported a vertical synchronization (VSYNC) interval.
INB$L_FREE_QUEUE_FLINK	The forward link to the next free packet in the free queue.
INB$L_FREE_QUEUE_BLINK	The backward link to the previous free packet in the free queue.
INB$L_NOHISTORY_FLINK	Reserved.
INB$L_NOHISTORY_BLINK	Reserved.
INB$L_HISTORY_BUFFER	A pointer to the motion history buffer (MHB).
INB$L_HISTORY_SIZE	The current size in pages of the motion history buffer.

(continued on next page)

Table A–2 (Cont.) Input Buffer Control Block Fields

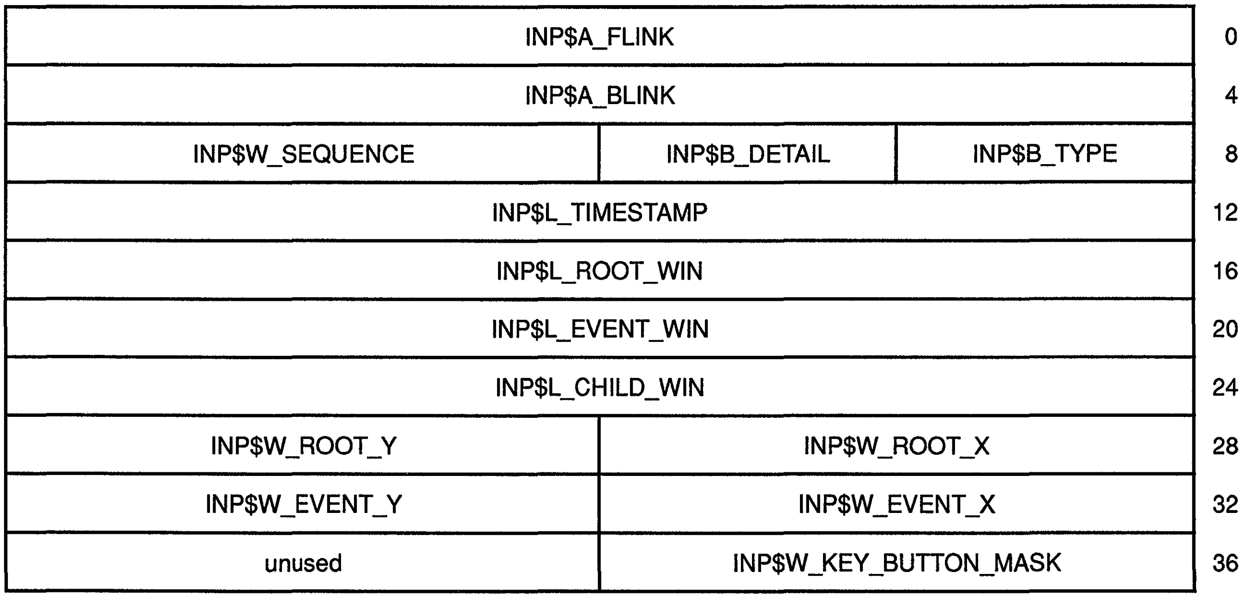
Field Name	Contents
INB$L_FLAGS	A 32-bit field containing interface status bits. Mask bits in this field correspond to five possible states: INB$V_ACB_BUSY — Bit 0, AST already queued. INB$V_AWAKE — Bit 1, AST not required for server. INB$V_SERVER_TIMED — Bit 2, if set, check server (AST-inhibit) timer for zero before sending AST. INB$V_NON_INT_BOX — Bit 3, if set, motion event data is inhibited when pointer is in the noninterest box. INB$V_MHB_BUSY — Bit 4, if set, the motion history buffer is busy with the server or is disabled by the server. A value of 0 enables the buffer.
INB$B_ACB	A pointer to the AST control block 28-byte array.
INB$L_SAVED_PID	The process ID of the device that sent the AST. EXE$QIO obtains the process identification from the PCB and writes the value into this field.
INB$W_NON_INT_BOX_X1	Address of the first *x*-axis pointer events noninterest box.
INB$W_NON_INT_BOX_Y1	Address of the first *y*-axis pointer events noninterest box.
INB$W_NON_INT_BOX_X2	Address of the second *x*-axis pointer events noninterest box.
INB$W_NON_INT_BOX_Y2	Address of the second *y*-axis pointer events noninterest box.
INB$B_SAVED_RMOD	The access mode value of the process at the time of the I/O request. EXE$QIO obtains the processor access mode from the PSL and writes the value into this field.
INB$W_VERSION	The INB version number.
INB$L_TIMESTAMP_MONTH	The timestamp with month.
INB$L_TIMESTAMP_MS	Timestamp to the nearest millisecond.
INB$L_SCHED_QUANTUM	Event time slice ir. server schedule.
INB$W_SCHED_FLAGS	A 16-bit field containing interface status bits concerning event scheduling in the server. Mask bits in this field correspond to three possible states: INB$V_INPUT_PENDING — Bit 0, packet in queue. INB$V_QUANTUM_ EXPIRED — Bit 1, time slot expired. INB$V_SCHED_YIELD — Bit 2, yield to a higher priority event.
INB$L_COUNTER1	Reserved.
INB$L_COUNTER2	Reserved.
INB$L_COUNTER3	Reserved.
INB$L_COUNTER4	Reserved.
INB$L_PFN_COUNT	A pointer to the page frame counter indicating the number of PFNs for the shared buffer.
INB$L_PFN_LIST	A pointer to the page frame number list.

A.3 Input Packet (INP)

The input packet (INP) data structure defines the packet format used in the interface between the common device driver and the DECwindows server. The basic DECwindows format of the input packet conforms with the X event format in the X Window System protocol.

Some fields in the packets of certain events may vary. The packet illustrated in Figure A–3 is a typical input event generated by a pointing device or keyboard. Input events include key, button, and pointer motion events. The first 12 bytes (three longwords) are common to all events. The event information is always 32 bytes, excluding the prefixed forward/backward pointers (FLINK/BLINK) for VMS. Table A–3 defines the fields of the packet.

Figure A–3 Input Packet Data (INP)



INP$A_FLINK			0
INP$A_BLINK			4
INP$W_SEQUENCE	INP$B_DETAIL	INP$B_TYPE	8
INP$L_TIMESTAMP			12
INP$L_ROOT_WIN			16
INP$L_EVENT_WIN			20
INP$L_CHILD_WIN			24
INP$W_ROOT_Y	INP$W_ROOT_X		28
INP$W_EVENT_Y	INP$W_EVENT_X		32
unused	INP$W_KEY_BUTTON_MASK		36

Table A–3 Input Packet Fields

Field Name	Contents
INP$L_FLINK	The forward link to the next INP structure. This field is filled in by the class driver queue routines.
INP$L_BLINK	The backward link to the previous INP structure. This field is filled in by the class driver queue routines.
INP$B_TYPE	Specifies the X11 event type. The class driver writes the packet type in this field when it creates the INP.

(continued on next page)

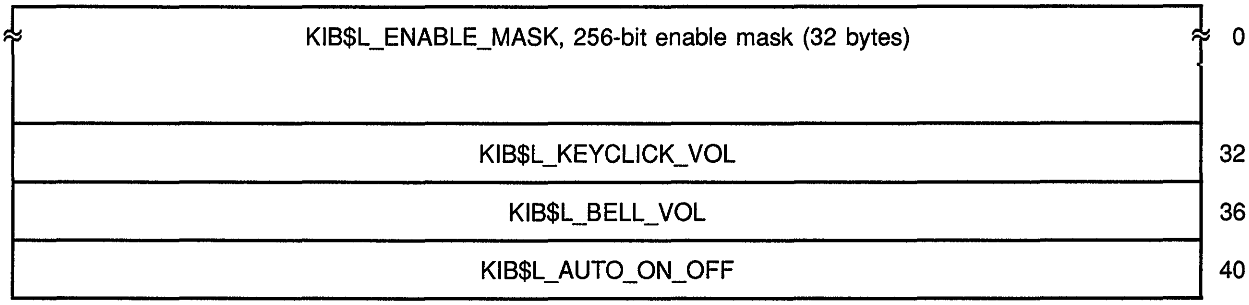
Table A–3 (Cont.) Input Packet Fields

Field Name	Contents	
INP$B_DETAIL	A keyboard/mouse event code specifying a key/button state. Possible key definition states: KEYPRESS, KEYRELEASE. Possible mouse definition states: BUTTONPRESS, BUTTONRELEASE, and MOTIONNOTIFY.	
INP$W_SEQUENCE	The packet sequence number in the transmission session.	
INP$L_TIMESTAMP	The date/time at packet creation.	
INP$L_ROOT_WIN	The pointer to a root window data structure in the display application.	
INP$L_EVENT_WIN	The pointer to the display window data structure in the application where the current activity is referenced.	
INP$L_CHILD_WIN	The pointer to a child display window in the application associated with the current activity. (The parent is the event window data structure.)	
INP$W_ROOT_X	A value in pixels specifying the horizontal placement of the upper left corner of the root window on the screen.	
INP$W_ROOT_Y	A value in pixels specifying the vertical placement of the upper left corner of the root window on the screen.	
INP$W_EVENT_X	A value in pixels specifying the pointer movement (left, right, distance) along the *x*-axis.	
INP$W_EVENT_Y	A value in pixels specifying the pointer movement (up, down, distance) along the *y*-axis.	
INP$W_KEY_BUTTON_MASK	A 16-bit mask marking the bit position that defines a specific function for a key or button. Bits in this field identify specific functions of the input event as follows:	
	INP$V_SHIFTMASK	Bit 0, shift key pressed.
	INP$V_CAPSLOCKMASK	Bit 1, caps lock key pressed.
	INP$V_CONTROLMASK	Bit 2, control key pressed.
	INP$V_MOD1MASK	Bit 3, the key pressed is modifying MB1.
	INP$V_MOD2MASK	Bit 4, the key pressed is modifying MB2.
	INP$V_MOD3MASK	Bit 5, the key pressed is modifying MB3.
	INP$V_MOD4MASK	Bit 6, the key pressed is modifying MB4.
	INP$V_MOD5MASK	Bit 7, the key pressed is modifying MB5.
	INP$V_BUTTON1MASK	Bit 8, MB1 is pressed.
	INP$V_BUTTON2MASK	Bit 9, MB2 is pressed.
	INP$V_BUTTON3MASK	Bit 10, MB3 is pressed.
	INP$V_BUTTON4MASK	Bit 11, MB4 is pressed.
	INP$V_BUTTON5MASK	Bit 12, MB5 is pressed.
	INP$V_UNUSED1MASK	Bit 13, reserved.
	INP$V_UNUSED2MASK	Bit 14, reserved.
	INP$V_ANYMODIFIER	Bit 15, set when any grab button or grab key modifier is pressed.

A.4 Keyboard Information Block (KIB)

The keyboard information block (KIB) in the I/O database contains keyboard characteristics that are used in the execution of the keyboard information $QIO system service. Data is passed between server and driver by the system service reading/writing the KIB during the keyboard information sense/set mode $QIO call. The **p2** $QIO parameter points to the starting address of the block. The fields define bell and keyclick volume and autorepeat information (see Figure A–4). Table A–4 lists and defines the fields of the block. The length of the data structure is defined by the constant KIB$S_KBD_INFO.

Figure A–4 Keyboard Information Block



KIB$L_ENABLE_MASK, 256-bit enable mask (32 bytes)	0
KIB$L_KEYCLICK_VOL	32
KIB$L_BELL_VOL	36
KIB$L_AUTO_ON_OFF	40

Table A–4 Keyboard Information Block Fields

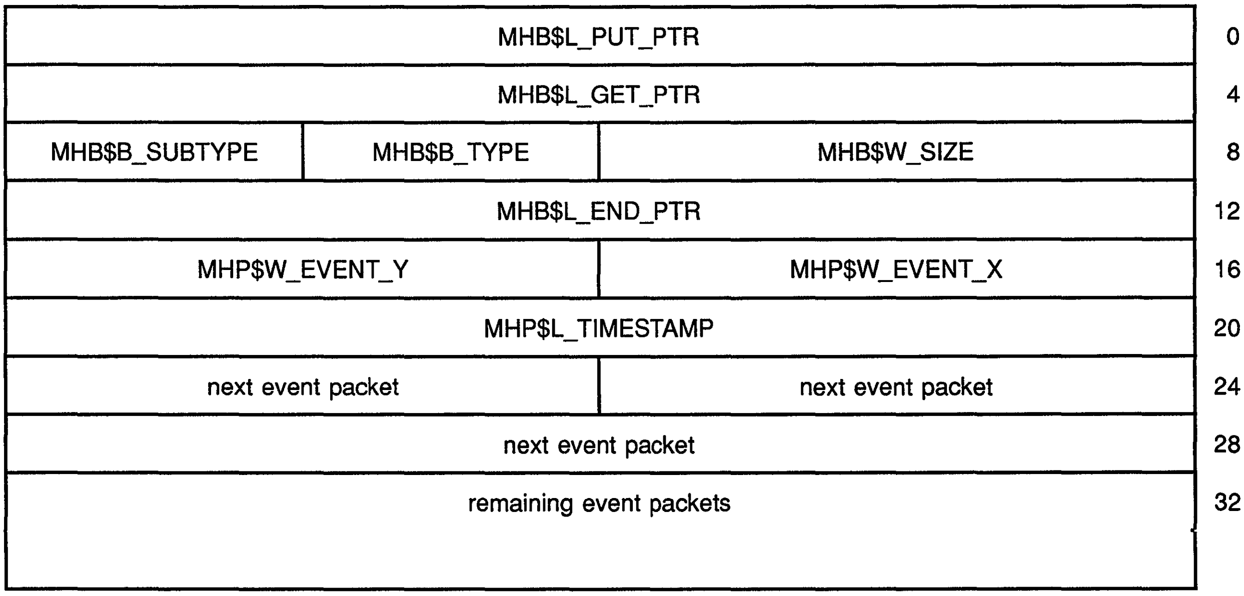
Field Name	Contents
KIB$L_ENABLE_MASK	Entry to the 256-bit autorepeat enable mask for the LK201 keys. The mask defines which keys are in autorepeat mode. The bits are numbered 0 through 255 and each bit position corresponds to a specific key position on an LK201 keyboard. For example, using decimal keycode numbering, mask-bit 90 corresponds to the 90 key position on the LK201 keyboard.
KIB$L_KEYCLICK_VOL	A longword specifying the keyclick volume in percent. A value of 100 specifies the loudest click while a 0 turns the keyclick off. A value of –1 provides a default volume of 70 percent.
KIB$L_BELL_VOL	A longword specifying the bell volume in percent. A value of 100 specifies the loudest ring while a 0 turns the bell off. A value of –1 provides a default volume of 70 percent.
KIB$L_AUTO_ON_OFF	Defines the state of the autorepeat feature. A value of 0 turns autorepeat on and a value of 1 turns it off.

A.5 Motion History Buffer (MHB)

The motion history buffer (MHB) data structure in the I/O database provides a storage area for pointing device movements as history events. The buffer structure contains a 16-byte control block or header at the top followed by (starting at address 16_{10}) 8-byte motion history packets (MHPs) that make up a ring buffer. Each history packet contains an *x*-axis and *y*-axis movement with an event timestamp (see Figure A–5). Table A–5 lists and defines the fields of the motion history buffer.

The MHB header and MHP packets are defined by the $DECWCOMMON macro. The MHB header length is defined by constant MHB$S_MHB_STRUCT and the MHP packet length (8 bytes) is defined by MHP$K_LENGTH. Field MHB$L_PUT_PTR points to the next free packet and field MHB$L_GET_PTR points to the oldest pointer motion event. Field MHB$L_END_PTR points to the last byte in the buffer.

Figure A–5 Motion History Buffer Data Structure



MHB$L_PUT_PTR			0
MHB$L_GET_PTR			4
MHB$B_SUBTYPE	MHB$B_TYPE	MHB$W_SIZE	8
MHB$L_END_PTR			12
MHP$W_EVENT_Y		MHP$W_EVENT_X	16
MHP$L_TIMESTAMP			20
next event packet		next event packet	24
next event packet			28
remaining event packets			32

Table A–5 Motion History Buffer Fields

Field Name	Contents
MHB$L_PUT_PTR	Points to the next or oldest free packet in the ring.
MHB$L_GET_PTR	Points to the oldest motion event packet in the ring.
MHB$W_SIZE	Size in bytes of the history buffer.

(continued on next page)

Table A–5 (Cont.) Motion History Buffer Fields

Field Name	Contents
MHB$B_TYPE	Defines the system or major type of data structure (DECwindows) that is read by the System Dump Analyzer (SDA). The common driver writes the symbolic constant DYN$C_DECW in this field when the common driver creates the MHB.
MHB$B_SUB_TYPE	Defines the specific (subtype) data structure (MHB) within the major type that is read by the SDA. The common driver writes the symbolic constant DYN$C_DECW_MHB in this field when the common driver creates the MHB.
MHB$L_END_PTR	Points to the last free packet in the ring.
MHB$T_RING	Starting address of the ring buffer packet area.
MHP$W_EVENT_X	A packet event value in pixels specifying the pointer movement (left, right, distance) along the *x*-axis.
MHP$W_EVENT_Y	A packet event value in pixels specifying the pointer movement (up, down, distance) along the *y*-axis.
MHP$L_TIMESTAMP	A packet event value specifying the date/time of the pointer movement.

A.6 Unit Control Block for Input Device

A unit control block (UCB) data structure is a variable-length block in the I/O database that describes the characteristics of a single device unit. The driver-loading procedure creates some static fields. The operating system and device drivers can read and modify all nonstatic fields of the UCB.

The general UCB structure for an input device with a port/class interface is shown in Figure A–6. It contains five sections: the system section (base UCB), the class driver terminal section, the DECwindows input device extension, the port driver terminal section, and the port extension region.

The system section of the terminal driver UCB contains the fields of the UCB that are present in all of the UCBs on the system. The length of the system UCB is defined by UCB$K_LENGTH.

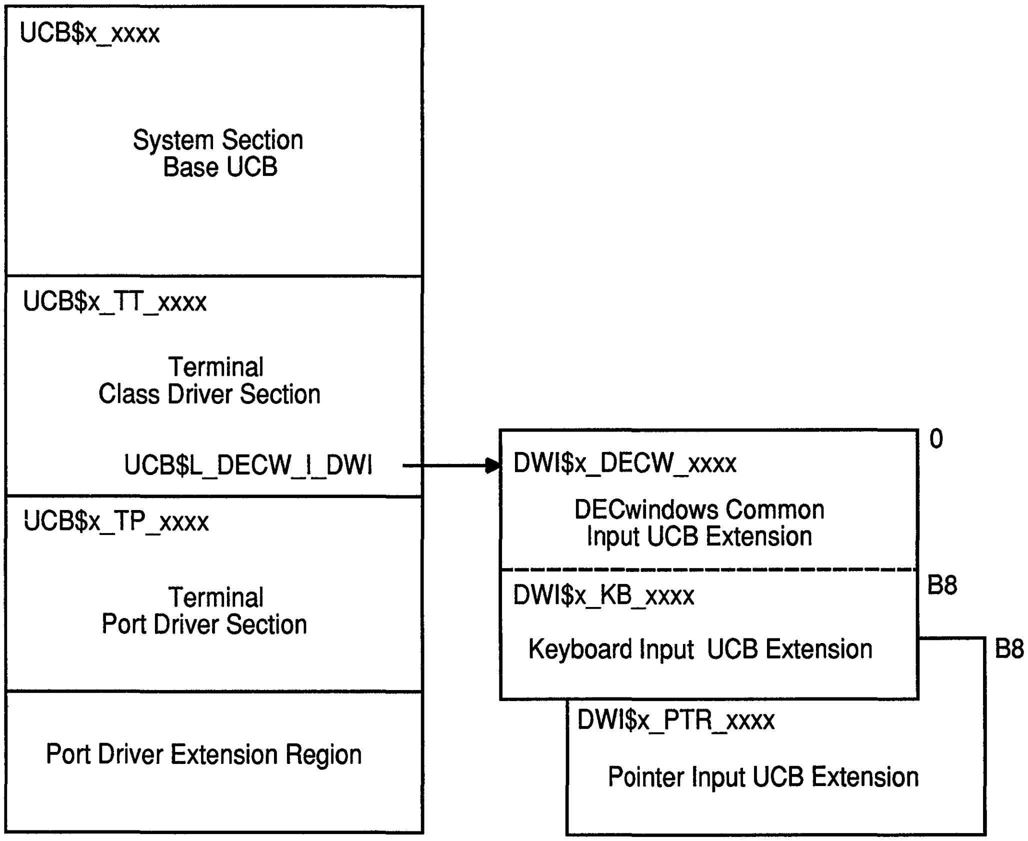
The class driver terminal section of the UCB contains fields that are needed by the class driver. These fields have names of the form UCB$*x*_TT_*fieldname*, where *x* denotes the field size and *fieldname* is the name of the field. The UCB$K_TT_LENGTH constant defines the length of the class driver section of the UCB.

The DECwindows input extension section contains fields that are needed by all DECwindows device drivers for communication and processing. These fields have names of the form DWI$*x*_*fieldname*, where *x* denotes the field size and *fieldname* is the name of the field. Symbol UCB$L_DECW_I_DWI is always the address of the input extension starting address.

The port driver terminal section of the UCB contains fields that both the class and port driver must access. These fields have names of the form UCB$*x*_TP_*fieldname*, where *x* denotes the field size and *fieldname* is the name of the field. Although a port driver may not actually use all these fields, they are needed by other software.

The terminal port extension region is defined by the terminal port driver. It can be any length and contain any context that the port driver needs in order to execute the port functions.

Figure A–6 Unit Control Block General Structure



ZK–0024A–GE

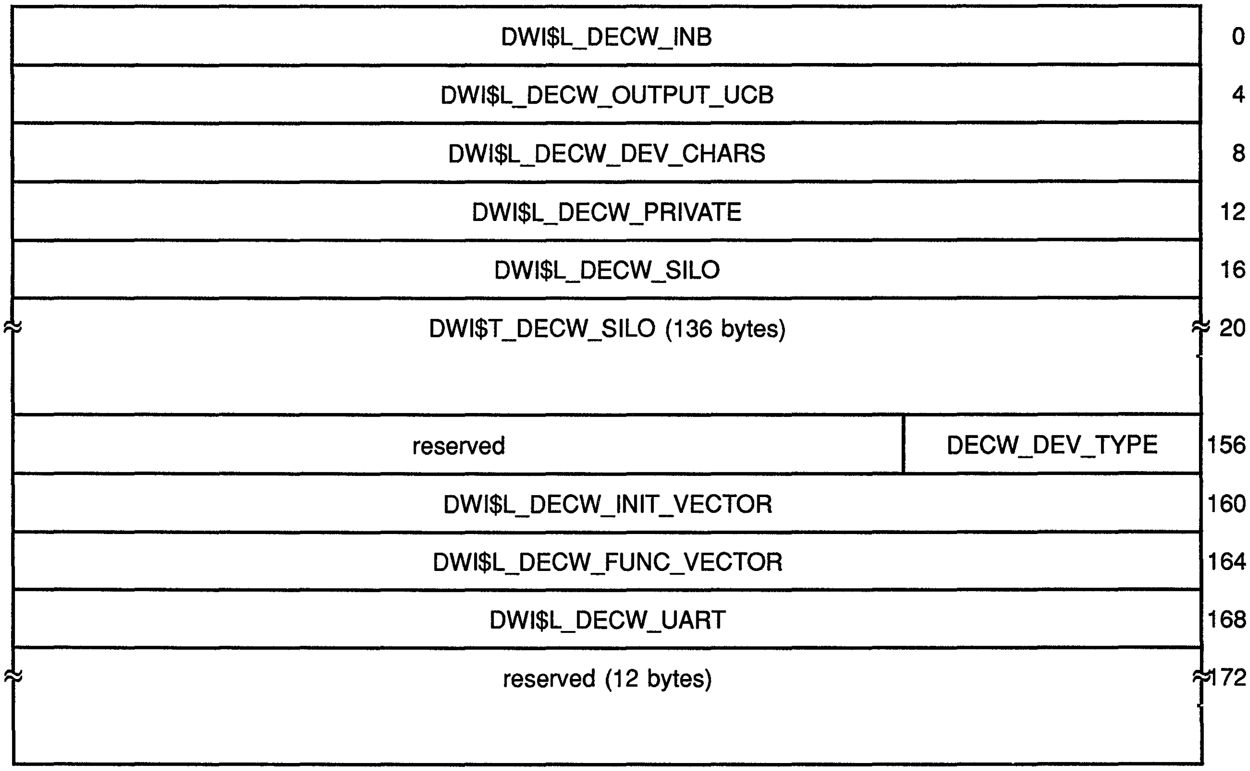
A.6.1 UCB/DECwindows Common Input Extension (DWI)

Each terminal device on the system has its own UCB, including VMS terminal extensions for port and class drivers (described in an appendix of the *VMS Device Support Manual*). A DECwindows UCB includes a common input extension (DWI/DECW) and a device-specific extension (see Figure A–6). UCB class driver section field UCB$L_DECW_I_DWI points to the DWI extension starting address. Note that field UCB$L_TT_WFLINK is overwritten and redefined as UCB$L_DECW_I_DWI when the

DECwindows extension is created by macro $DECWINPUTUCB (invoked by $DECWGBL). The common DWI extension length is specified at B8$_{16}$ (184$_{10}$) by DWI$K_DECW_COMMON_LENGTH.

This section describes the DWI common input extension structure (see Figure A–7). Table A–6 lists and defines the fields of the DECwindows UCB common input extension.

Figure A–7 UCB/DECwindows Common Input Extension



DWI$L_DECW_INB		0
DWI$L_DECW_OUTPUT_UCB		4
DWI$L_DECW_DEV_CHARS		8
DWI$L_DECW_PRIVATE		12
DWI$L_DECW_SILO		16
DWI$T_DECW_SILO (136 bytes)		20
reserved	DECW_DEV_TYPE	156
DWI$L_DECW_INIT_VECTOR		160
DWI$L_DECW_FUNC_VECTOR		164
DWI$L_DECW_UART		168
reserved (12 bytes)		172

Table A–6 UCB/DECwindows Common Input Extension Fields

Field Name	Contents
DWI$L_DECW_INB	Pointer to the INB header.
DWI$L_DECW_OUTPUT_UCB	Pointer to the start of the output UCB.
DWI$L_DECW_DEV_CHARS	Pointer to the DVI.
DWI$L_DECW_PRIVATE	Defines the escape point for extensions (reserved for DIGITAL).

(continued on next page)

Table A–6 (Cont.) UCB/DECwindows Common Input Extension Fields

Field Name	Contents
DWI$L_DECW_SILO	Pointer to the start of the SILO[1] block.
DWI$T_DECW_SILO	The SILO buffer area (block) defined by the $SILODEF macro.
DWI$B_DECW_DEV_TYPE	A byte that defines the input device type (mouse/keyboard).
DWI$L_DECW_INIT_VECTOR	Offset to the initialization routine that starts the device self test.
DWI$L_DECW_FUNC_VECTOR	Offset to the powerfail function routine.
DWI$L_DECW_UART	Address of the UART/CSR[2] for the serial input device.

[1] SILO is a software version of a Service In Logical Order buffer for serial input and output data in the channel associated with the UCB.

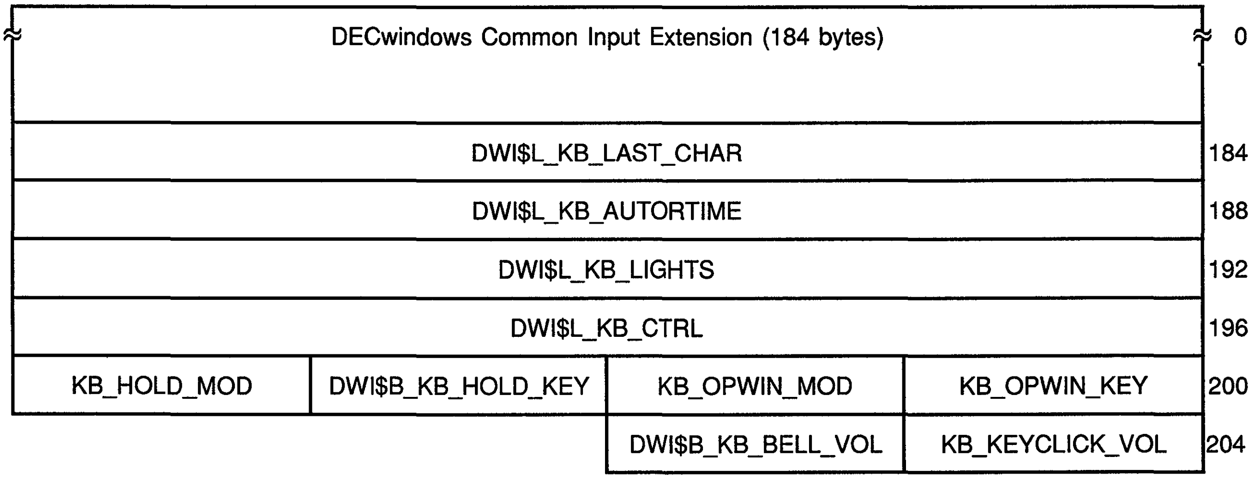
[2] UART is a universal asynchronous receiver/transmitter chip for serial interfaces and CSR is the Control/Status Register.

A.6.2 UCB/DECwindows Keyboard Input Extension (DWI)

Each input device requires a specific block of information that is contiguous with the end of the common UCB input extension. The specific device extension information starts at address $B8_{16}$ (184_{10}). This section describes the DWI keyboard input extension structure that is shown in Figure A–8 starting at DWI$L_KB_LAST_CHAR ($184_{10}$). Table A–7 lists and defines the fields of the DECwindows UCB keyboard input extension.

UCB class driver section field UCB$L_DECW_I_DWI points to the DWI common input extension starting address. Note that field UCB$L_TT_WFLINK is overwritten and redefined as UCB$L_DECW_I_DWI when macro $DECWINPUTUCB creates the DECwindows extension. The common DWI extension length is specified as $B8_{16}$ (184_{10}) by DWI$K_DECW_COMMON_LENGTH and the keyboard extension length is specified as 114_{16} (276_{10}) by DWI$K_KB_LENGTH.

Figure A–8 UCB/DECwindows Keyboard Input Extension



DECwindows Common Input Extension (184 bytes)				0
DWI$L_KB_LAST_CHAR				184
DWI$L_KB_AUTORTIME				188
DWI$L_KB_LIGHTS				192
DWI$L_KB_CTRL				196
KB_HOLD_MOD	DWI$B_KB_HOLD_KEY	KB_OPWIN_MOD	KB_OPWIN_KEY	200
		DWI$B_KB_BELL_VOL	KB_KEYCLICK_VOL	204

(continued on next page)

Figure A–8 (Cont.) UCB/DECwindows Keyboard Input Extension

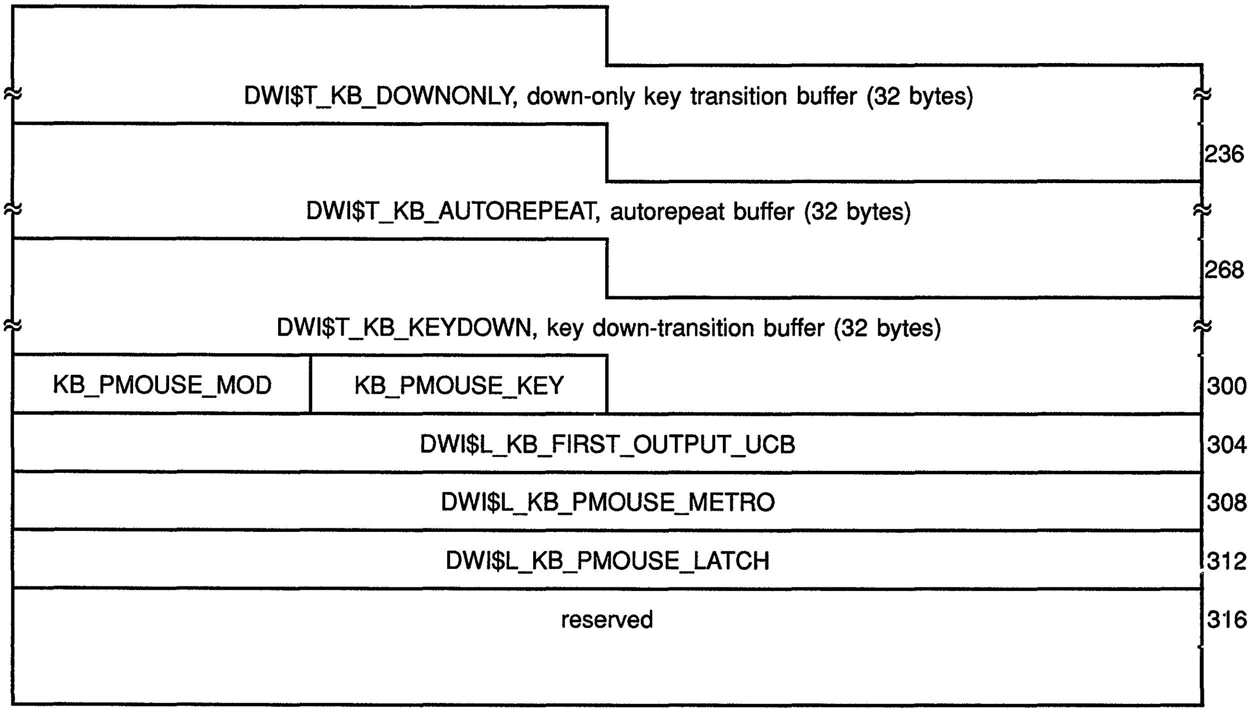

DWI$T_KB_DOWNONLY, down-only key transition buffer (32 bytes)
236
DWI$T_KB_AUTOREPEAT, autorepeat buffer (32 bytes)
268
DWI$T_KB_KEYDOWN, key down-transition buffer (32 bytes)
KB_PMOUSE_MOD
KB_PMOUSE_KEY
300
DWI$L_KB_FIRST_OUTPUT_UCB
304
DWI$L_KB_PMOUSE_METRO
308
DWI$L_KB_PMOUSE_LATCH
312
reserved
316

Table A–7 UCB/DECwindows Keyboard Input Extension Fields

Field Name	Contents	
DWI$L_KB_LAST_CHAR	Defines the last input character.	
DWI$L_KB_AUTORTIME	Defines the autorepeat time (metronome) in milliseconds.	
DWI$L_KB_LIGHTS	Defines the keyboard lights (LEDs).	
DWI$L_KB_CTRL	Contains the status input flags using macro $VIELD from the controller CSR. Mask flag bits in this field correspond to ten possible states:	
	FLAG$V_SHIFT	Bit 0 defines the state of the Shift key.
	FLAG$V_LOCK	Bit 1 defines the state of the Lock key.
	FLAG$V_CTRL	Bit 2 defines the state of the Ctrl key.
	FLAG$V_BUTTONDOWN	Bit 3 indicates that a mouse button down transition occurred.
	FLAG$V_LOCKDOWN	Bit 4 defines the up/down state of the lock key
	FLAG$V_AUTOREPEAT	Bit 5 indicates that the software autorepeat timer is enabled/disabled.

(continued on next page)

Table A–7 (Cont.) UCB/DECwindows Keyboard Input Extension Fields

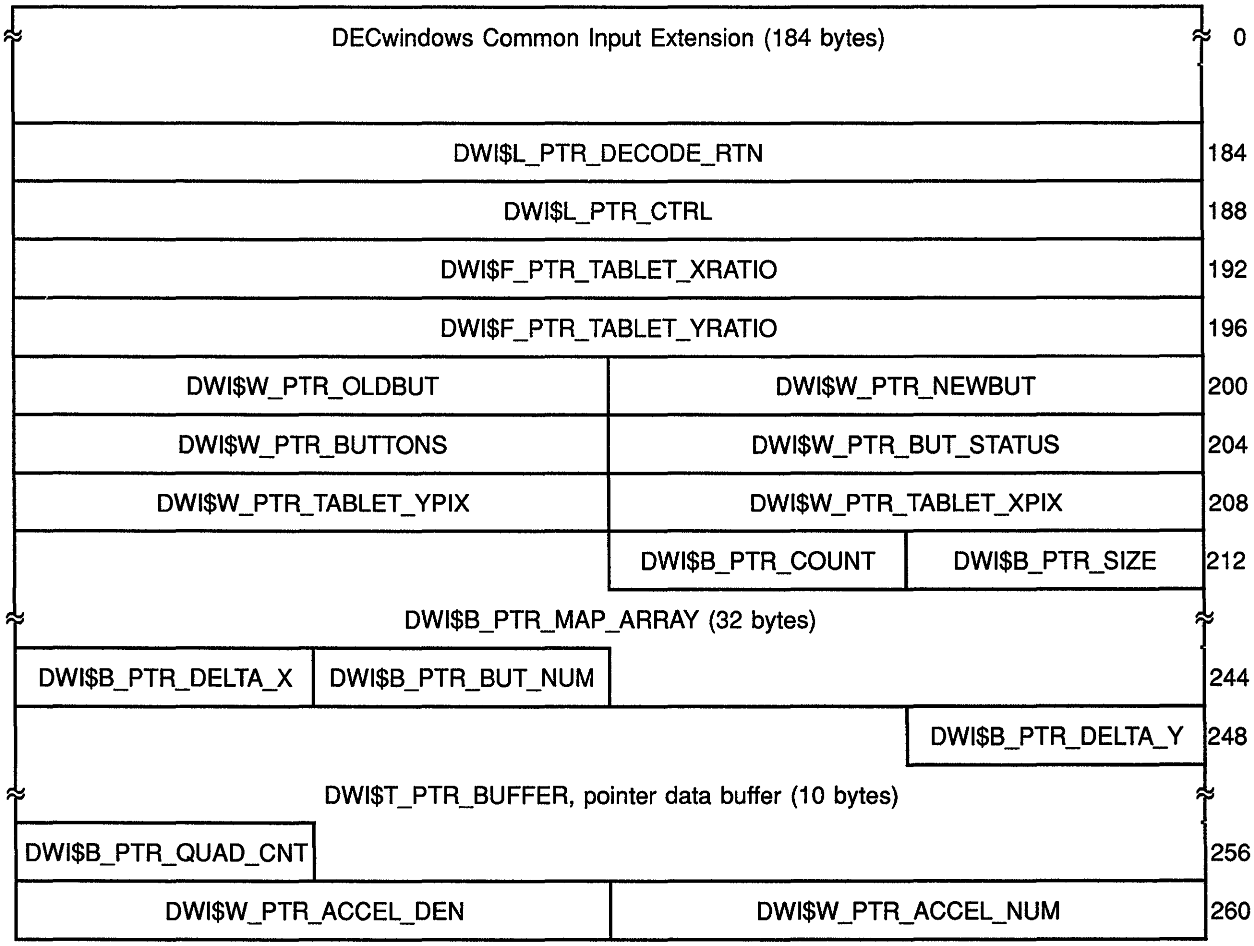
Field Name	Contents	
	FLAG$V_UART_KEYBOARD	Bit 6 indicates that the serial port (UART) for the keyboard is enabled/disabled.
	FLAG$V_PMOUSE	Bit 7 indicates that the pseudomouse mode is enabled/disabled.
	FLAG$V_AUTOREPEAT_OFF	Bit 8 is the global autorepeat flag. A value of 0 indicates the autorepeat mode, a value of 1 indicates the up/down mode with no keys in autorepeat.
	FLAG$V_MAIN_KB_UPDOWN	Bit 9 indicates the up/down mode for the main keyboard.
DWI$B_KB_OPWIN_KEY	Specifies the primary key to invoke the operator window.	
DWI$B_KB_OPWIN_MOD	Defines the modifier key (Ctrl or Shift) used with the primary key to invoke the operator window.	
DWI$B_KB_HOLD_KEY	Specifies the primary key to invoke the hold-screen mode.	
DWI$B_KB_HOLD_MOD	Defines the modifier key (Ctrl or Shift) used with the primary key to invoke the hold-screen mode.	
DWI$B_KB_KEYCLICK_VOL	Defines the keyclick volume.	
DWI$B_KB_BELL_VOL	Defines the bell volume.	
DWI$T_KB_DOWNONLY	Starting address of the 32-byte down-only key transition buffer for all LK201 key codes.	
DWI$T_KB_AUTOREPEAT	Starting address of the software autorepeat flag buffer for all LK201 key codes.	
DWI$T_KB_KEYDOWN	Starting address of the 32-byte current key down transition buffer for all LK201 key codes.	
DWI$B_KB_PMOUSE_KEY	Defines the primary key to invoke the pseudomouse mode.	
DWI$B_KB_PMOUSE_MOD	Defines the modifier key (Ctrl or Shift) used with the primary key to invoke pseudomouse mode.	
DWI$L_FIRST_OUTPUT_UCB	Defines the first output device connected.	
DWI$L_KB_PMOUSE_METRO	Counter for the autorepeat pseudomouse.	
DWI$L_KB_PMOUSE_LATCH	Defines the keyboard keys for the pseudomouse buttons/latches. To simulate a button hold down for the keyboard user, a latch key is provided with each mouse-button key. Within the longword, four bytes define the latch keys corresponding to the simulated mouse buttons as follows:	
	DWI$B_PMOUSE_LATCH1	Defines the keyboard key (raw key code) to latch the MB1 key.
	DWI$B_PMOUSE_LATCH2	Defines the keyboard key (raw key code) to latch the MB2 key.
	DWI$B_PMOUSE_LATCH3	Defines the keyboard key (raw key code) to latch the MB3 key.
	DWI$B_PMOUSE_LATCH_SAVE	Defines the keyboard key (raw key code) that is currently latched.

A.6.3 UCB/DECwindows Pointer Input Extension (DWI)

Each input device requires a specific block of information that is contiguous with the end of the common UCB input extension. The specific device extension information starts at address B8$_{16}$ (184$_{10}$). This section describes the DWI pointer input extension structure for mouse device information that is shown in Figure A–9 starting at DWI$L_PTR_DECODE_RTN (184$_{10}$). Table A–8 lists and defines the fields of the DECwindows UCB pointer input extension.

UCB class driver section field UCB$L_DECW_I_DWI points to the DWI common input extension starting address. Note that field UCB$L_TT_WFLINK is overwritten and redefined as UCB$L_DECW_I_DWI when macro $DECWINPUTUCB creates the DECwindows extension. The common DWI extension length is specified as B8$_{16}$ (184$_{10}$) by DWI$K_DECW_COMMON_LENGTH and the pointer extension length is specified as 11C$_{16}$ (284$_{10}$) by DWI$K_PTR_LENGTH.

Figure A–9 UCB/DECwindows Pointer Input Extension



(continued on next page)

Figure A–9 (Cont.) UCB/DECwindows Pointer Input Extension

reserved	DWI$W_PTR_ACCEL_THR	264
DWI$L_PTR_MOTION_COMP_HIT		268
reserved		272

Table A–8 UCB/DECwindows Pointer Input Extension Fields

Field Name	Contents
DWI$L_PTR_DECODE_RTN	Points to the specific decoding routine for the serial data.
DWI$L_PTR_CTRL	Contains the pointing device status input flags by way of macro $VIELD from the controller CSR. Mask flag bits in this field correspond to five possible states: FLAG$V_QUAD_MOUSE — Bit 0 specifies that a QV mouse[1] is connected. FLAG$V_SERIAL_MOUSE — Bit 1 specifies that a serial mouse is connected. FLAG$V_SERIAL_TABLET — Bit 2 specifies that a serial tablet is connected. FLAG$V_STYLUS — Bit 3 specifies that a stylus is connected. FLAG$V_MOTION_COMP — Bit 4 indicates the pointer motion compression mode.
DWI$F_PTR_TABLET_XRATIO	Pointer to the tablet *x*-ratio array.
DWI$F_PTR_TABLET_YRATIO	Pointer to the tablet *y*-ratio array.
DWI$W_PTR_NEWBUT	Contains the new button status.
DWI$W_PTR_OLDBUT	Contains the old button status.
DWI$W_PTR_BUT_STATUS	Current button status.
DWI$W_PTR_BUTTONS	Button status in raw format.
DWI$W_PTR_TABLET_XPIX	Defines the tablet/stylus *x* position.
DWI$W_PTR_TABLET_YPIX	Defines the tablet/stylus *y* position.
DWI$B_PTR_SIZE	Defines the number of bytes for every pointer report.
DWI$B_PTR_COUNT	Defines the byte count of the pointer data buffer.
DWI$B_PTR_MAP_ARRAY	Starting address of the 32-byte pointer mapping array.
DWI$B_BUT_NUM	Defines the number of buttons mapped.
DWI$B_PTR_DELTA_X	Defines the change in *x* position.

[1] QV mouse is an older version pointing device (model VS10X-EA).

(continued on next page)

Table A–8 (Cont.) UCB/DECwindows Pointer Input Extension Fields

Field Name	Contents
DWI$B_PTR_DELTA_Y	Defines the change in *y* position.
DWI$T_PTR_BUFFER	Starting address of the 10-byte pointer data buffer.
DWI$B_PTR_QUAD_CNT	Contains the byte count for a QV mouse.[1]
DWI$B_PTR_KEYBOARD_UCB	Points to the keyboard UCB when there is no output device.
DWI$W_PTR_ACCEL_NUM	Defines the pointer acceleration table numerator.
DWI$W_PTR_ACCEL_DEN	Defines the pointer acceleration table denominator.
DWI$W_PTR_ACCEL_THR	Defines the pointer acceleration threshold.
DWI$L_PTR_MOTION_COMP_HIT	Defines the count of motion compression hits that indicates the number of lost pointer-motion events stored in the motion history buffer.

[1] QV mouse is an older version pointing device (model VS10X-EA).

A.7 Unit Control Block for Output Device

A unit control block (UCB) data structure is a variable-length block in the I/O database that describes the characteristics of a single device. The driver loading procedure creates some static fields. VMS and device drivers can read and modify all nonstatic fields of the UCB.

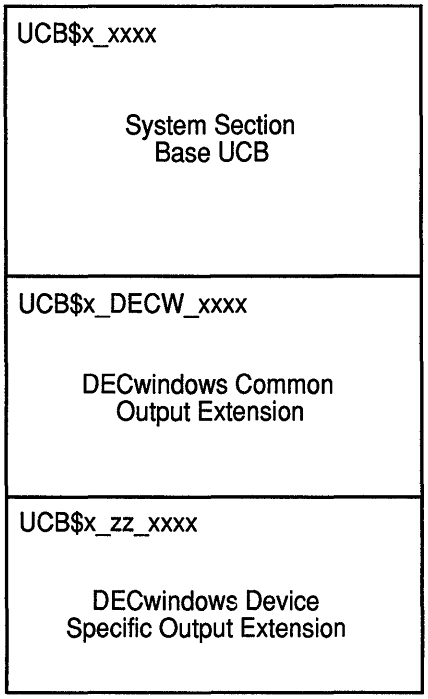
The general UCB structure for an output device for DECwindows is shown in Figure A–10. It contains three sections: the system section (base UCB), the DECwindows common output extension, and the DECwindows output device-specific extension.

The system section of the terminal driver UCB contains the fields of the UCB that are present in all of the UCBs on the system. The length of the system UCB is defined by UCB$K_LENGTH.

The common output extension of the UCB contains fields that are required by all DECwindows output drivers. These fields have names of the form UCB$*x*_DECW_*fieldname*, where *x* denotes the field size and *fieldname* is the name of the field. The UCB$K_DECW_COMMON_LENGTH constant defines the length of the common output extension.

The output-device-specific extension of the UCB contains fields that both the output driver and the device must access. These fields have names of the form UCB$*x_zz_fieldname*, where *x* denotes the field size, where *zz* sometimes implies the controller type, and *fieldname* is the name of the field.

Figure A–10 Unit Control Block Output Device General Structure



ZK–0025A–GE

A.7.1 UCB/DECwindows Common Output Extension (UCB/DECW)

A DECwindows output device UCB includes a DECwindows common output extension (DECW) to the main system UCB. This section describes the structure of the UCB/DECW common output extension (see Figure A–11). The main system UCB is described in an appendix of the *VMS Device Support Manual.* Table A–9 lists and defines the fields of the DECwindows UCB common output extension.

The output extension is created by macro $DECWOUTPUTUCB (invoked by $DECWGBL) and follows the system base UCB starting at symbolic offset UCB$K_LENGTH. The output common extension length is defined by UCB$K_DECW_COMMON_LENGTH.

Figure A–11 UCB/DECwindows Common Output Extension

UCB$L_DECW_OUTPUT_VECTOR	0
UCB$L_DECW_DVI	4

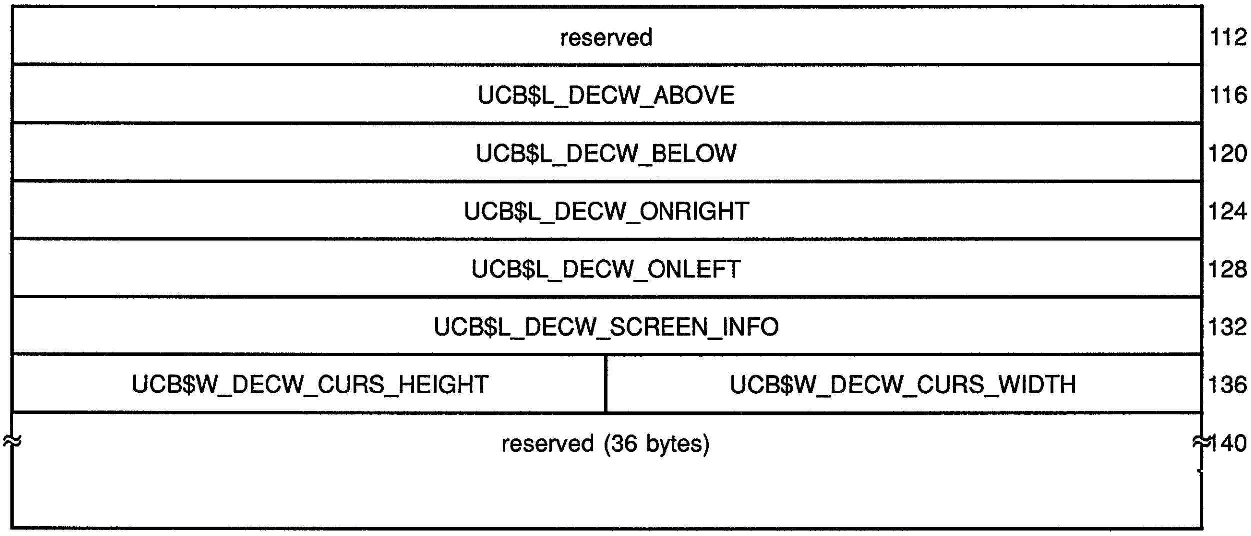
(continued on next page)

Figure A–11 (Cont.) UCB/DECwindows Common Output Extension

UCB$L_DECW_KB_UCB				8
UCB$L_DECW_PTR_UCB				12
UCB$L_DECW_CSR				16
UCB$L_DECW_TIMER				20
UCB$L_DECW_SCRSAV_TIMEOUT				24
UCB$L_DECW_CTRL				28
UCB$L_DECW_CURSOR_PATTERN				32
UCB$L_DECW_WAIT_FLINK				36
UCB$L_DECW_WAIT_BLINK				40
UCB$L_DECW_VSYNC_INTERVAL				44
UCB$L_DECW_HW_PTR_UCB				48
UCB$L_DECW_HW_KB_UCB				52
UCB$L_DECW_ATTACHED_FLINK				56
UCB$L_DECW_ATTACHED_BLINK				60
UCB$W_DECW_SNIFF_STYLE		UCB$W_DECW_NEG_VSYNC_MS		64
UCB$W_DECW_CURSOR_X		UCB$W_DECW_SNIFF_CYCLE		68
UCB$W_DECW_CURSOR_XOFF		UCB$W_DECW_CURSOR_Y		72
UCB$W_DECW_MAX_X		UCB$W_DECW_CURSOR_YOFF		76
UCB$W_DECW_MAXCURS_X		UCB$W_DECW_MAX_Y		80
UCB$W_DECW_SCRSAV_CYCLE		UCB$W_DECW_MAXCURS_Y		84
UCB$W_DECW_RED_CURSOR		UCB$W_DECW_CURSOR_LENGTH		88
red hotspot		red background		92
green background		UCB$W_DECW_GREEN_CURSOR		96
UCB$W_DECW_BLUE_CURSOR		green hotspot		100
blue hotspot		blue background		104
reserved	DECW_SCRSAV_LIGHTS	DECW_LAST_Y	UCB$B_DECW_LAST_X	108

(continued on next page)

Figure A-11 (Cont.) UCB/DECwindows Common Output Extension



reserved		112
UCB$L_DECW_ABOVE		116
UCB$L_DECW_BELOW		120
UCB$L_DECW_ONRIGHT		124
UCB$L_DECW_ONLEFT		128
UCB$L_DECW_SCREEN_INFO		132
UCB$W_DECW_CURS_HEIGHT	UCB$W_DECW_CURS_WIDTH	136
reserved (36 bytes)		140

Table A-9 UCB/DECwindows Common Output Extension Fields

Field Name	Contents	
UCB$L_DECW_OUTPUT_VECTOR	Address of the output vector table.	
UCB$L_DECW_DVI	Pointer to the DVI.	
UCB$L_DECW_KB_UCB	Address of the keyboard UCB.	
UCB$L_DECW_PTR_UCB	Address of the mouse UCB.	
UCB$L_DECW_CSR	Address of the monitor Control and Status Register (CSR).	
UCB$L_DECW_TIMER	Contains the vertical synchronization (VSYNC) timer value (16.6 milliseconds to zero).	
UCB$L_DECW_SCRSAV_TIMEOUT	Defines the screen saver timeout period.	
UCB$L_DECW_CTRL	Defines the control/status field for workstation configuration information. Bits in this longword field specify the workstation configuration as follows:	
	FLAG$V_SCRSAV_ON	Bit 0, set if screen saver is active.
	FLAG$V_RESET_SCRSAV	Bit 1 is set to reset screen saver.
	FLAG$V_RELOAD_DONE	Bit 2, set when driver is reloaded.
	FLAG$V_CONSOLE	Bit 3, set when console is available for this device.
	FLAG$V_OPWIN_ACTIVE	Bit 4, set when the operator window is active. (Check operator reference count for 0 and expected F2 key.)

(continued on next page)

Table A–9 (Cont.) UCB/DECwindows Common Output Extension Fields

Field Name	Contents	
	FLAG$V_CTRL_INIT_ SUCCESS	Bit 5, set when controller initialization completes successfully.
	FLAG$V_CURSOR_OFF_ SCREEN	Bit 6, set if cursor is on another screen.
UCB$L_DECW_CURSOR_PATTERN	Address of cursor pattern bits.	
UCB$L_DECW_WAIT_FLINK	Defines the forward link in the IRP wait list (during first system initialization only).	
UCB$L_DECW_WAIT_BLINK	Defines the backward link in the IRP wait list (during first system initialization only).	
UCB$L_DECW_VSYNC_INTERVAL	Defines the VSYNC interval in milliseconds.	
UCB$L_DECW_HW_PTR_UCB	Address of the pointing device UCB on this workstation.	
UCB$L_DECW_HW_KB_UCB	Address of the keyboard UCB on this workstation.	
UCB$L_DECW_ATTACHED_FLINK	Defines the forward link in the list of attached screens.	
UCB$L_DECW_ATTACHED_BLINK	Defines the backward link in the list of attached screens.	
UCB$W_DECW_NEG_VSYNC_MS	Contains the negated VSYNC interval in milliseconds.	
UCB$W_DECW_SNIFF_STYLE	Defines the cursor style that is checked at every sniff cycle. A value of 0 defines a black cursor, a value of 1 defines a white cursor, and a value of 2 defines a two-plane (black outlined) cursor.	
UCB$W_DECW_SNIFF_CYCLE	The sniff cycle defines the interval (the number of vertical SYNCs for VAXstation II, black and white only) at which the hotspot and cursor characteristics are checked. Typically, the cursor characteristics are checked on every tenth vertical SYNC. A value of 10 selects every tenth vertical SYNC as the sniff cycle.	
UCB$W_DECW_CURSOR_X	Defines the current cursor *x* position.	
UCB$W_DECW_CURSOR_Y	Defines the current cursor *y* position.	
UCB$W_DECW_CURSOR_XOFF	Defines the current cursor *x* position offset.	
UCB$W_DECW_CURSOR_YOFF	Defines the current cursor *y* position offset.	
UCB$W_DECW_MAX_X	Defines the maximum *x* position of the screen (minus one).	
UCB$W_DECW_MAX_Y	Defines the maximum *y* position of the screen (minus one).	
UCB$W_DECW_MAXCURS_X	Address of the maximum *x*-coordinate of the cursor.	
UCB$W_DECW_MAXCURS_Y	Address of the maximum *y*-coordinate of the cursor.	
UCB$W_DECW_SCRSAV_CYCLE	Defines the screen saver cycle count.	
UCB$W_DECW_CURSOR_LENGTH	Defines the cursor pattern length.	
UCB$W_DECW_RED_CURSOR	Three words defining the red foreground, red background, and hotspot.	
UCB$W_DECW_GREEN_CURSOR	Three words defining the green foreground, background, and hotspot.	
UCB$W_DECW_BLUE_CURSOR	Three words defining the blue foreground, background, and hotspot.	
UCB$B_DECW_LAST_X	Defines the previous cursor *x* position.	
UCB$B_DECW_LAST_Y	Defines the previous cursor *y* position.	
UCB$B_DECW_SCRSAV_LIGHTS	Defines the active keyboard lights during screen save.	

(continued on next page)

Table A–9 (Cont.) UCB/DECwindows Common Output Extension Fields

Field Name	Contents
UCB$L_DECW_ABOVE	UCB of screen above this one.
UCB$L_DECW_BELOW	UCB of screen below this one.
UCB$L_DECW_ONRIGHT	UCB of screen to the right of this one.
UCB$L_DECW_ONLEFT	UCB of screen to the left of this one.
UCB$L_DECW_SCREEN_INFO	Defines the screen information passed to the server.
UCB$W_DECW_CURS_WIDTH	Defines the cursor width boundary for hotspot location checking.
UCB$W_DECW_CURS_HEIGHT	Defines the cursor height boundary for hotspot location checking.

A.8 Class Vector Table

The class vector table (IK$VECTOR or IM$VECTOR) data structure contains vectors to the routines of the class driver module (see Figure A–12). The driver builds the data structure using the $VECINI, $VEC, and $VECEND macros. Table A–10 lists and defines the fields of the data structure.

Figure A–12 Class Vector Table Data Structure

CLASS_PUTNXT	0
CLASS_GETNXT	4
reserved	8
reserved	12
CLASS_DDT	16
reserved	20

Table A–10 Class Vector Table Fields

Field Name	Contents
CLASS_PUTNXT	Points to the put-next-input-data-on-queue routine (IK$PUTNXT) in the class input driver module.
CLASS_GETNXT	Points to the get-next-byte-for-output routine (IK$GETNXT) in the class input driver module.
CLASS_DDT	Points to the class driver dispatch table.

A.9 Common Vector Table

The common vector table (DECW_COMMON_VECTOR) data structure in the I/O database contains vectors to the routines and data segments of the common driver module (see Figure A–13). The driver builds the data structure using the $VECINI, $VEC, and $VECEND macros. Table A–11 lists and defines the fields of the data structure.

Figure A–13 Common Vector Table Data Structure

COMMON_DDT	0
COMMON_POS_CURSOR	4
COMMON_SETUP_INPUT_UCB	8
COMMON_SETUP_OUTPUT_UCB	12
COMMON_VSYNC	16
reserved	20
COMMON_FLAGS	24
reserved	28

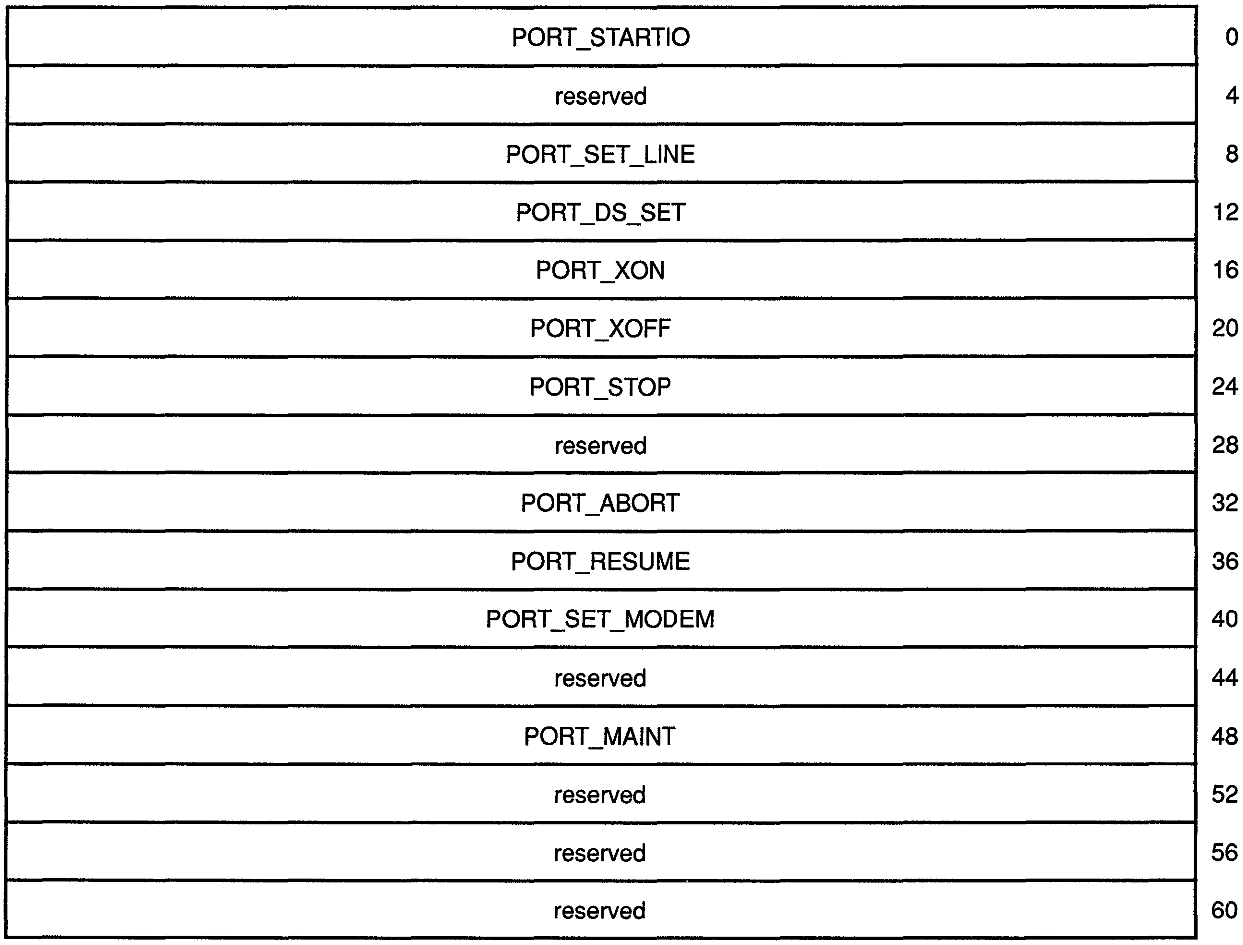
Table A–11 Common Vector Table Fields

Field Name	Contents
COMMON_DDT	Pointer to the driver dispatch table (IN$DDT).
COMMON_POS_CURSOR	Points to the cursor positioning routine (IN$POS_CURSOR).
COMMON_SETUP_INPUT_UCB	Points to the UCB setup routine (IN$SETUP_INPUT_UCB) for the input drivers.
COMMON_SETUP_OUTPUT_UCB	Points to the UCB setup routine (IN$SETUP_OUTPUT_UCB) for the output driver.
COMMON_VSYNC	Points to the vertical retrace timer routine (IN$VSYNC) for a time mark in the common driver code.
COMMON_FLAGS	Points to the flags longword in the common driver that stores global status.

A.10 Port Vector Table

The port vector table (PORT_VECTOR) data structure in the I/O database contains vectors to the routines of the input port driver module (see Figure A–14). The driver builds the data structure using the $VECINI, $VEC, and $VECEND macros. Table A–12 lists and defines the fields of the data structure.

Figure A–14 Port Vector Table Data Structure



PORT_STARTIO	0
reserved	4
PORT_SET_LINE	8
PORT_DS_SET	12
PORT_XON	16
PORT_XOFF	20
PORT_STOP	24
reserved	28
PORT_ABORT	32
PORT_RESUME	36
PORT_SET_MODEM	40
reserved	44
PORT_MAINT	48
reserved	52
reserved	56
reserved	60

Table A–12 Port Vector Table Fields

Field Name	Contents
PORT_STARTIO	Vector to the start I/O routine in the port input driver module.
PORT_SET_LINE	Points to the set-terminal-line-characteristics routine in the port input driver module.
PORT_DS_SET	Points to the set-modem-output-signals routine in the port input driver module.
PORT_XOFF	Points to the send XOFF routine in the port input driver module.

(continued on next page)

Table A–12 (Cont.) Port Vector Table Fields

Field Name	Contents
PORT_XON	Points to the send XON routine in the port input driver module.
PORT_STOP	Points to the reset-active-output routine in the port input driver module.
PORT_ABORT	Points to the abort-active-output routine in the port input driver module.
PORT_RESUME	Points to the resume-stopped-output routine in the port input driver module.
PORT_SET_MODEM	Points to the initialize-modem-polling routine in the port input driver module.
PORT_MAINT	Points to the DZ11 maintenance routine in the port input driver module.

A.11 Output Vector Table

The output vector table (GA$VECTOR or GC$VECTOR) data structure contains vectors to the routines of the output driver module. The output driver builds the data structure using the $VECINI, $VEC, and $VECEND macros. Table A–13 lists and defines the fields of the data structure.

Table A–13 Output Vector Table Fields

Field Name	Contents
OUTPUT_CLEAR_CURSOR	Points to the clear-cursor routine (CLEAR_CURSOR) in the output driver subroutine module (not used in GPX drivers).
OUTPUT_CURSOR_PATTERN	Points to the set- or load-cursor-pattern routine in the output driver subroutine module.
OUTPUT_DISABLE_VIDEO	Points to the disable-video routine (DISABLE_VIDEO) in the main output driver module.
OUTPUT_ENABLE_VIDEO	Points to the enable-video routine (ENABLE_VIDEO) in the main output driver module.
OUTPUT_BUFFERED_FDT	Points to the FDT-parsing routine (GA$FDTPARSE) in the main GPX output driver module.
OUTPUT_POS_CURSOR	Points to the position-cursor routine in the output driver subroutine module.
OUTPUT_CANCEL	Points to the cancel-I/O routine (GA$CANCEL) in the GPX main output driver module.
OUTPUT_DIRECT_FDT	Points to the FDT-parsing routine (GA$FDTPARSE) in the main GPX output driver module.
OUTPUT_SET_DVI	Points to the set- or initialize-the-DVI-data-structure routine in the output driver subroutine module.

(continued on next page)

Table A–13 (Cont.) Output Vector Table Fields

Field Name	Contents
OUTPUT_OPWIN_VISIBLE	Points to the check-for-operator-window-mode-capability routine in the output driver subroutine module.
OUTPUT_OPWIN_UP	Points to the display-operator-window routine in the output driver subroutine module.
OUTPUT_OPWIN_DOWN	Points to the remove-operator-window routine in the output driver subroutine module.
OUTPUT_OPWIN_RESIZE	Points to the resize-operator-window routine in the output driver subroutine module.

B Device Driver Macros

Macros within device driver software modules ensure consistency and simplify the coding of the DECwindows interface. This appendix describes the macros used in the various driver modules. They are presented in three groups:

- General device driver
- Input queue and packet processing
- Vector generation

B.1 General Device Driver Macros

This section describes the VMS macros that provide general services within the device driver modules. Information concerning data structures referenced in this chapter may be found in Appendix A. The general device driver macros are as follows:

- COMMON_CTRL_INIT
- COMMON_UNIT_INIT
- $DECW_COMMON_READY
- $DECWGBL

COMMON_CTRL_INIT

The COMMON_CTRL_INIT macro relocates a vector table generated by the $VEC macro. The controller initialization routine containing this macro is called at system startup and during recovery after power failure.

FORMAT

COMMON_CTRL_INIT *dpt, vector*

arguments

dpt is a symbolic name of the driver prologue table.

vector is the address of the table generated by $VEC.

output

Location	Contents
R0	Destroyed
R1	Destroyed

COMMON_UNIT_INIT

The COMMON_UNIT_INIT macro sets the driver dispatch table field in the device data block and UCB to contain the address of the common DDT. This macro contains the common code for the unit initialization routine that is run whenever a unit is created.

FORMAT

COMMON_UNIT_INIT

input

Location	Contents
R0	Address of the DDT for this unit
R5	UCB address

output

Location	Contents
R1	Destroyed
R2	Destroyed

$DECW_COMMON_READY

The $DECW_COMMON_READY macro senses whether the common drivers are loaded. The macro is used by all DECwindows drivers that depend on a common driver to operate. It checks for the presence of the common driver (INDRIVER) by examining the symbol DECW$GL_VECTOR, set by the driver's controller initialization routine. It then checks for keyboard and mouse class drivers by looking at the flags longword (indexed from the common vector).

FORMAT $DECW_COMMON_READY

output

Location	Contents
R0	Contains a value of 0 when the common driver is not loaded. Contains a value of 1 when all common drivers are loaded.

$DECWGBL

The $DECWGBL macro is an external definition that defines the DECwindows common, input, and output driver data structures. The global macro directly calls other structure definition macros within the various DECwindows and VMS modules (see Table B–1) that define the I/O database.

Table B–1 Structure Definition Macros Called by $DECWGBL

Macro Name	Macro Function
$TTYVECDEF	Defines the port and class vector tables.
$SILODEF	Defines the SILO block.
$DECWDEF	Defines the DVI, KIB, and INP data structures.
$DECWCMNINPUCB	Defines the common input extension (DWI$x_DECW_) and calls $TTYUCBDEF to link the device-specific input extension structure.
$DECWPTRINPUCB	Defines the pointing device input UCB extension (DWI$x_ PTR_).
$DECWKBINPUCB	Defines the keyboard input UCB extension (DWI$x_KB_).
$DECWOUTPUTUCB	Defines the output device UCB extension (UCB$x_ DECW_).
$DECWCOMMON	Defines the INB and MHB, the common vector and output vector tables, common flag word (COMMON_FLAGS), and calls the dynamic definition macro $DYNDEF.

B.2 Input Queue and Packet Processing Macros

This section describes the macros that are used in a class input driver module to acquire free input packets and insert them into the input queue. The input queue and packet-processing macros are as follows:

- GET_FREE_KB_PACKET
- GET_LAST_EVENT_PACKET
- PUT_INPUT_ON_QUEUE

GET_FREE_KB_PACKET

The GET_FREE_KB_PACKET macro returns the address of a free packet in R1 if one is available. If a packet is not available, 0 is returned in R1. It uses the interlocked queue instruction (REMQHI) to remove the packet from the free queue.

FORMAT GET_FREE_KB_PACKET

input

Location	Contents
R5	UCB address of the input device

output

Location	Contents
R1	Address of the free packet; 0 if there is none
R2	Destroyed

GET_LAST_EVENT_PACKET

The GET_LAST_EVENT_PACKET macro removes a specified type event packet from the tail of the input queue for motion compression. The macro tests the packet type (INP$B_TYPE) to ensure that it matches the **event_type** argument in the macro call. If a match is found, the macro increments the motion compression count (DWI$L_PTR_MOTION_COMP_HIT) and returns the free-queue address of the removed input packet in R1. If an **event_type** (typically, X$MOTION_NOTIFY) packet is not found, it reinserts the packet in the input queue and returns a 0 in R1.

FORMAT

GET_LAST_EVENT_PACKET *event_type*

arguments

event_type defines the X11 event (X$MOTION_NOTIFY) for which the packet is removed from the input queue for compression.

input

Location	Contents
R5	UCB address of the input device

output

Location	Contents
R1	Address of the removed packet; 0 if there is none
R2	Destroyed
DWI$L_PTR_MOTION_ COMP_HIT	Motion compression count, incremented by one if the specified event type packet is found

PUT_INPUT_ON_QUEUE

The PUT_INPUT_ON_QUEUE macro inserts an input packet onto the input queue shared with the server. The macro uses the interlocked queue instruction INSQTI for queue insertion and uses the queue header information from the UCB to select the right input queue.

FORMAT PUT_INPUT_ON_QUEUE

input

Location	Contents
R1	Address of the input event packet
R5	UCB address of the input device

output

Location	Contents
R2	Destroyed

B.3 Vector Table Generation Macros

This section describes the VMS macros that should be used to generate the various vector tables. Using these macros ensures the generation of a valid table, even if the vector table is expanded in new releases. The vector table generation macros are as follows:

- $VECINI
- $VEC
- $VECEND

$VECINI

The $VECINI macro generates and initializes the vector table. The table is initialized with the entries pointing to the driver's null routine. Subsequent calls to $VEC fill the table with the addresses of the real entry points.

FORMAT

$VECINI *drivername, null_routine, [prefix]*

arguments

drivername defines the driver prefix, usually two alphabetic characters.

null_routine defines the address of the driver's null entry point.

prefix defines the prefix to be added to the generated symbols (for instance, PORT_, CLASS_, COMMON_, OUTPUT_). PORT_ is the default.

Note: **The null routine should simply contain an RSB instruction. It is called for any function that the driver does not support.**

$VEC

The $VEC macro generates fields (vectors) and validates the vector table entry. Each invocation of the $VEC macro specifies the **entry** argument. However, a driver need not supply the address of a routine for each entry in the table. The $VEC macro constructs a valid table regardless of how many entries are supplied. The $VEC macro accepts the entry names minus the driver type prefix (PORT_ or CLASS_ or OUTPUT_). (For examples, refer to the figures for the class or port vector table data structures in Appendix A.) The $VECINI macro defines the prefix applied to the entries: PORT_ for the port vector table and CLASS_ for the class vector table. This macro ensures that a working table is generated, or that you are notified of any error by message. Note that a driver accesses the table using the symbolic offset names shown in the vector table data structures of Appendix A.

FORMAT

$VEC *entry, routine*

arguments

entry defines the name of the table entry.

routine defines the name of the routine being inserted in the entry.

$VECEND

The $VECEND macro generates the longword of zeros that terminates the vector table list and sets the location counter to the correct position. The exact placement of $VECEND in the sequence of $VEC entries marks the end of the vector table.

FORMAT

$VECEND *[end]*

arguments

end is a flag controlling the generation of the end of the vector table. This argument is generally omitted so that the $VECEND macro can generate the end of the vector table. Otherwise, the $VECEND macro does not generate the end of the table.

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	**Call**	**Contact**
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local DIGITAL subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local DIGITAL subsidiary or approved distributor
Internal[1]	————	SDC Order Processing - WMO/E15 *or* Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

[1] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS DECwindows Device Driver Manual
AA–MG28A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page Description

_____ ______________________________

_____ ______________________________

_____ ______________________________

_____ ______________________________

_____ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** _____ of the software this manual describes.

Name/Title ______________________________ Dept. __________

Company ______________________________ Date __________

Mailing Address ______________________________

______________________________ Phone __________

— Do Not Tear - Fold Here and Tape

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

VMS DECwindows Device Driver Manual
AA–MG28A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page Description

______ ______________________________

______ ______________________________

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________

______________________________ Phone ______________

— Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here

Cut Along Dotted Line