94-003111501

MicroVMS
Workstation Video Device Driver Manual

Order Number: AA-DY65D-TE
and Update Notice No. 1
(AD-DY65D-T1)
and Update Notice No. 2
(AD-DY65D-T2)

July 1987

The *MicroVMS Workstation Video Device Driver Manual* provides technical information on the operations of the QVSS and QDSS drivers.

Operating System and Version: MicroVMS Version 4.5

Software Version: MicroVMS Workstation Version 3.2

digital equipment corporation
maynard, massachusetts

July 1987

The READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

S790

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Update Notice 2

MicroVMS Workstation Video Device Driver Manual

Order Number: AD-DY65D-T2

July 1987

NEW INFORMATION

This update contains changes to the *MicroVMS Workstation Video Device Driver Manual,* AA-DY65D-TE.

digital equipment corporation
maynard, massachusetts

INSTRUCTIONS

The enclosed pages are replacements for current pages of the *MicroVMS Workstation Video Device Driver Manual.* On replacement pages, changes and additions are indicated by vertical bars (|).

Keep this notice in your manual to maintain an up-to-date record of changes.

Old Page	New Page
Title/Copyright	Title/Copyright
iii/iv through xiii/blank	iii/iv through xiii/blank
1–5/1–6 through 1–7/1–8	1–5/1–6 through 1–8.1/blank
3–3/3–4 through 3–7/3–8	3–3/3–4 through 3–7/3–8
4–31/4–32	4–31/4–32
5–19/5–20 through 5–89/5–90	5–19/5–20 through 5–89/5–90
A–11/A–12 through A–15/A–16	A–11/A–12 through A–15/A–16
Index–1/Index–2 through Index–5/Index–6	Index–1/Index–2 through Index–7/blank
Reader's Comments/Mailer	Reader's Comments/Mailer

Update Notice No. 1

MicroVMS Workstation Video Device Driver Manual

Order Number: AD–DY65D–T1

November 1986

New and Changed Information

This update contains changes and additions made to the *MicroVMS Workstation Video Device Driver Manual.*

Instructions

The enclosed pages are to be placed in the *MicroVMS Workstation Video Device Driver Manual* as replacements for or additions to current pages. A change made on a replacement page is indicated in the right-hand margin by a change bar (▮). If a page has a date at the top, but no change bars, the text on that page is revised.

Old Page	New Page
Title page/Copyright page	Title page/Copyright page
iii/iv through ix/blank	iii/iv through ix/blank
2-59/2-60	2-59/blank
3-1/3-2 through 3-67/blank	3-1/3-2 through 3-65/3-66
4-1/4-2 through 4-37/4-38	4-1/4-2 through 4-39/blank
Index-1/Index-2 through Index-5/Index-6	Index-1/Index-2 through Index-5/Index-6
Reader's Comments/Mailer	Reader's Comments/Mailer
Reader's Comments/Mailer	Reader's Comments/Mailer

Contents

Chapter 2 How to Program to the Driver

Chapter 3 QVSS/QDSS Common QIO Interface

Chapter 4 QDSS-Specific QIO Interface

Chapter 5 Using Drawing Operation Primitives

Examples

Figures

Tables

Preface

The *MicroVMS Workstation Video Device Driver Manual* provides a programmer with the necessary information for writing applications that manipulate the QVSS and QDSS drivers.

It is structured to serve as both a tutorial manual that will bring an experienced programmer up to speed on driver concepts and as a reference manual that can be used for quick reference during actual application programming.

QIOs and system routines used when programming to the driver are provided in reference form. Each data type used to program to the drivers is illustrated in the reference sections when referred to and all the data types are summarized in two data-type appendixes.

Intended Audience

The information contained in this manual is for experienced graphics programmers or system programmers who are writing applications directly to the driver.

Structure of This Document

This manual has the following structure:

- Chapter 1 describes concepts and terms needed to understand programming to the driver interface.
- Chapter 2 describes how to perform driver interface tasks that are common to both the QVSS and QDSS systems.
- Chapter 3 describes the common QVSS/QDSS QIO interface.
- Chapter 4 describes the QDSS-specific QIO interface.
- Chapter 5 describes how to use the Drawing Operation Primitive interface.
- Appendix A describes all QVSS/QDSS common data types.
- Appendix B describes all QDSS-specific data types.

- Appendix C describes all multiplane writing modes.
- Appendix D contains a full QVSS driver example.
- Appendix E contains macros used to construct keyboards.
- Appendix F contains macros used to construct compose tables.
- Appendix G contains the default three-stroke compose table macros.
- Appendix H contains the $QIO system service description.
- Appendix I contains the DEC multinational character set table.

Associated Documents

The following MicroVMS manuals are related to this manual:

- *MicroVMS Workstation Graphics Programming Guide*
- *MicroVMS Workstation User's Guide*
- *MicroVMS User's Guide*

Conventions Used in This Document

This manual uses the following conventions in displaying the syntax requirements of user input to the system and in displaying examples:

Conventions	Meaning
RETURN key	The RETURN key is not always shown in formats and examples. Assume that you must press RETURN after typing a command or other input to the system unless instructed otherwise.
CTRL key	The word CTRL followed by a slash followed by a letter means that you must type the letter while holding down the CTRL key. For example, CTRL/B means hold down the CTRL key and type the letter B.

Conventions	Meaning
Lists	When a format item is followed by a comma and an ellipsis (, . . .), you can enter a single item or a number of those items separated by commas. When a format item is followed by a plus sign and an ellipsis (+ . . .), you can enter a single item or a number of those items connected by plus signs. If you enter a list (more than one item), you must enclose the list in parentheses. A single item need not be enclosed in parentheses.
Optional items	An item enclosed in square brackets ([ ]) is optional.
Key Symbols	In examples, keys and key sequences appear as symbols such as PF2 and CTRL/Z .
Ellipsis . . .	A vertical ellipsis indicates that some of the format or example is not included.
Delete Key	The key on the VT200 series terminal keyboard that performs the DELETE function is labeled <X . Assume that DELETE in text and examples refers to both the VT100 and VT200 series delete keys.
Examples	Examples show both system output (prompts, messages, and displays) and user input. User input is printed in red.

New and Changed Features

The following changes have been made for Version 3.0 of the MicroVMS Workstation software:

- The QDSS driver is available — only on systems with QDSS hardware. The QDSS driver permits you to draw multiplane (color) images through the use of the hardware-assisted Drawing Operation Primitive (DOP) interface. The QDSS system also uses a QIO interface. Read Chapters 1 and 2 for an overview of the driver.
- New QDSS-specific QIOs — see Chapter 4.
- New DOP interface — see Chapter 5.
- New UISDC routines for use with the DOP interface — see Chapter 5.

Chapter 1

Video Device Driver Introduction

This chapter provides an introduction to the concepts and terms this manual uses to describe how to write an application that interacts with the QVSS and QDSS video device drivers. Once you have read this chapter, you will know which programming interface your application should address, and you will be ready to proceed to the task-oriented Chapter 2.

This chapter describes:

- An overview of the two available video device drivers (QVSS and QDSS).
- How you determine which programming interface your application should address. (Your application may not need to write to a device driver.)
- How the two drivers address the screen.
- How the drivers use memory.
- How an application accesses a driver.

Some of the concepts and terms described in the following sections apply to both drivers, while others are specific to one driver. The manual clearly notes sections that describe specific concepts.

1.1 Overview of the Drivers

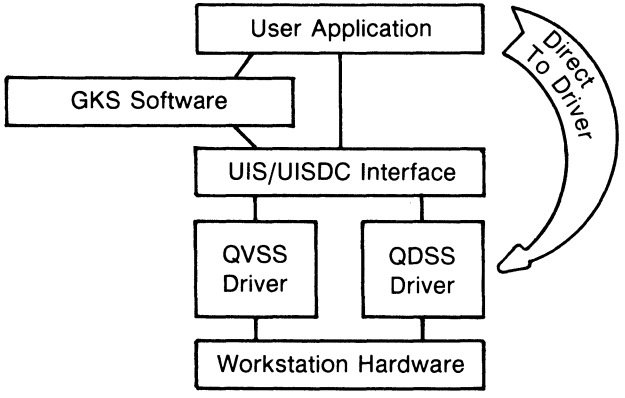
A VAXstation may have one of the following two video device drivers:

- The Q-Bus Video SubSystem (referred to as QVSS)
- The Q-Bus Device SubSystem (referred to as QDSS)

Both drivers allow you to create graphics applications that use the VAXstation features. However, each driver requires unique hardware, so a VAXstation can be configured with either a QVSS driver or a QDSS driver, but not both.

The device drivers provide a common interface to VAXstation hardware functions such as manipulating memory and writing to the screen. By using a common interface, applications can guarantee that the hardware is accessed in a uniform fashion. All VAXstation software uses the driver to access the hardware, either directly or indirectly. Figure 1–1 illustrates the layered relationship of applications, VAXstation software, the device driver, and VAXstation hardware.

Figure 1-1 The Driver Interface



1.1.1 Driver Differences

There are a number of differences between the QVSS driver and the QDSS driver, the primary ones being the use of color, the method of bitmap manipulation, and the ability to provide alternate windowing systems.

Color

The QVSS device driver is designed for use with a one-plane memory system. It is therefore restricted to the use of black and white images.

The QDSS device driver is designed for use with multiplane memory systems, so the QDSS driver has the capability to draw color and gray-scale images.

Bitmap Manipulation

The QVSS driver only supports direct bitmap manipulation. If you write an application to the QVSS driver, your application is completely responsible for drawing to the bitmap.

The QDSS driver provides a number of drawing operations that make drawing to the bitmap easier and faster. These operations are drawing operation primitives (referred to as DOPs) and are described in detail in Chapter 5. DOPs use additional multiplane hardware to accelerate drawing.

Alternate Windowing

The QVSS driver provides a way to alternate between the UIS windowing system and a windowing system of your own design. Both windowing systems share video memory when you use alternate windowing systems.

The QDSS driver does **not** currently provide a way to alternate between windowing systems; it supports only one windowing system at any given time.

1.2 Which Interface to Choose

When writing an application, you must determine which level of interface your application should address. As shown in Figure 1–1, your application may address the system at the UIS level, the driver level, or even the hardware level. Each successive level downward increases the degree of control your application has while increasing the amount of work your application must perform.

UIS and UISDC Interface

UIS, the VAXstation graphics operating system, provides a basic set of graphic primitive, color, and windowing routines to use when writing high-level graphics applications. UIS routines use a world coordinate system.

If your application requires direct access to display coordinates, it can use the UISDC routines. The UISDC routines allow you to manipulate primitives in a device-dependent manner. The UIS and UISDC routines are described in *The MicroVMS Workstation Graphics Programming Guide.*

If the UIS interface provides all the necessary functionality for your application, address the UIS interface.

Driver Interface

Your application can bypass the UIS/UISDC interface and manipulate the driver directly. This feature allows you to create graphics packages tailored to your specific needs. For example, you can design your own windowing system, or you can provide your own drawing routines.

This manual provides the information necessary for an application to access either the QVSS or QDSS driver directly. The manual is intended for system programmers with a working knowledge of basic graphic concepts.

Hardware Interface

It is also possible to bypass both the UIS interface and the driver interface and directly address the hardware. If your application has no need for a windowing system, you may wish to consider writing directly to the hardware. How to address the hardware directly is beyond the scope of this manual. Refer to the hardware documentation for information about the hardware interface.

1.3 How the Drivers Work

This section contains a general description of how the drivers work. It introduces many concepts and terms that must be understood before you can attempt to write to the driver interface.

1.3.1 Defining Regions on the Screen

Both drivers address rectangular portions of the screen referred to as *regions*. The QDSS driver accesses regions of the screen as *viewports* (see Section 1.3.10 for more information on viewports). Your application defines the addressable regions of the screen using QIOs that are part of the driver interface.

For the driver to define a region, your application must associate the region with a unique channel. The channel provides a logical path connecting the application to the device driver. To obtain a unique channel number for a region, call the $ASSIGN system service before you define the region.

When your application defines a region, it associates the region with one of the following *events*:

- Cursor pattern
- Keyboard input
- Pointer button transition
- Pointer movement
- Viewport (QDSS only)

The QIO you use to define the region determines the type of event the driver associates with the region. For example, if you want the driver to associate a region with button transitions, you should define the region with the Enable Button Transition QIO.

When you define a region and associate it with an event, you can also specify an action to take place when the event is detected in that region by specifying the address of an AST action routine as a parameter of the region-defining QIO. For

example, you can define a region, associate it with a button transition, and specify that the region be erased when the driver detects the specified transition.

It is the job of the driver to detect when an event occurs and to ensure that the proper action takes place when it does. Section 1.3.4 describes how the drivers manage regions and events.

1.3.2 Driver Communication Mechanisms

An application can access either driver, using a QIO interface.

An application may also access the QDSS driver, using a mechanism referred to as the request queue. The request queue interface is QDSS specific.

1.3.2.1 The QIO Interface

The QIO interface is the method of access that is common to most drivers. The QVSS and QDSS drivers provide a number of QIOs that perform driver-specific functions, such as initializing the screen, defining a pointer movement region, or performing a bitmap copy.

While the QIO interface is common to both drivers, some QIOs are QDSS-specific. Any driver QIO can be used with the QDSS driver, while the QIOs that apply to both the QVSS and QDSS drivers are a subset of the entire QIO interface. The structure of this manual reflects this fact: Chapter 3 describes in detail the subset of QIOs that apply to both the QVSS and QDSS drivers; Chapter 4 describes the QIOs that apply only to the QDSS driver.

The majority of the QIOs are input functions. They are typically called with IO$_SETMODE specified as the function parameter. In these QIOs, the **P1** parameter actually serves as the distinguishing function code.

The remaining QIOs are output functions. They are typically called with IO$_SENSEMODE specified as the function parameter. Again, in these QIOs, the **P1** parameter actually serves as the distinguishing function code.

While the QIO interface permits a wide range of functions, some may be grouped together in functional categories. The following sections contain general descriptions of those categories. (See Chapters 3 and 4 for complete descriptions of *all* QIOs.)

Tracking Associated Events and Regions

Several input QIOs permit an application to construct *list entries*. The QVSS and QDSS drivers keep track of which regions are associated with which events by keeping lists — one list for each type of event. Typically, when you define a region, you use an input QIO that passes the driver the following information:

- Region description
- Associated event

- Address of an AST routine that defines the action to take when the event occurs

The driver uses this information to construct a list entry, which it places on the appropriate list. When an event occurs, the driver searches the lists and triggers the AST pointed to by the appropriate list entry. Section 1.3.4 elaborates on how the driver constructs and manages lists.

Returning System Information

The output QIOs permit your application to get information from the system. The Get System Information QIO returns the system information block, which describes the state of the system such as:

- Dimensions (and subdivisions) of video memory
- Current pointer position
- Current button status

The system information block differs slightly depending on whether you are on a QVSS system or a QDSS system. The QDSS system block contains all the same fields as the QVSS system block, but includes additional fields. Appendixes A and B illustrate and describe the structures of both system blocks. (On the QVSS system, the information block is referred to as the *QVB*; on the QDSS system, it is referred to as the *QDB*.)

You may also inquire about the characteristics of the current keyboard, using the Get Keyboard Characteristics QIO.

On QDSS systems only, you can obtain a viewport ID for use in subsequent operations, as well as color map information. See Section 1.3.10 for information about viewports.

Queue Manipulation

This section applies only to QDSS systems.

Several QDSS-specific output QIOs are used for manipulating the QDSS-specific queues: the request queue, the return queue, and the deferred queue. These QIOs permit an application to stop and start processing on the queues. The request queue is briefly described in the next section. A full description of the three queues and their interaction appears at the end of this chapter.

1.3.2.2 The Request Queue Interface

This section applies only to QDSS systems.

The request queue interface allows your application to perform drawing operations and to manipulate queues. The request queue interface involves less overhead than the QIO interface.

To use the request queue, your application must use drawing operation primitives (DOPs). A DOP is a data structure that contains the data needed to perform a drawing operation. Your application submits DOPs to the request queue for execution. The request queue is a double-linked list of all the DOPs waiting for execution. The order in which the DOPs are placed on the queue is the order in which the DOPs are executed (drawing is performed).

Section 1.3.11 describes the QDSS-specific queues in detail. Chapter 5 describes how to use DOPs.

1.3.3 How the Drivers Use Memory

To write to the screen, an application actually writes to video memory. The driver then maps the video memory to the screen. Understanding how the drivers use memory is essential to understanding how to program to the drivers.

1.3.3.1 How QVSS Uses Video Memory

The VAXstation display area is 1024 x 864 pixels. The full QVSS video memory is a 1024 x 2048-bit block of memory or, to correspond to the screen, 2048 lines of 1024 pixels. QVSS uses a data structure referred to as the *scanline map* to map lines of video memory to lines on the screen.

The scanline map is a contiguous-word index into video memory, which indexes the entire screen display area (864 lines). Each entry in the scanline map is a 0-relative, 16-bit word value that functions as an index into video memory. The first entry in the scanline map locates the first line of the screen display, the second entry contains the second line, and so forth. For each bit that is set on an indexed line of video memory, the driver sets a corresponding pixel on the display screen.

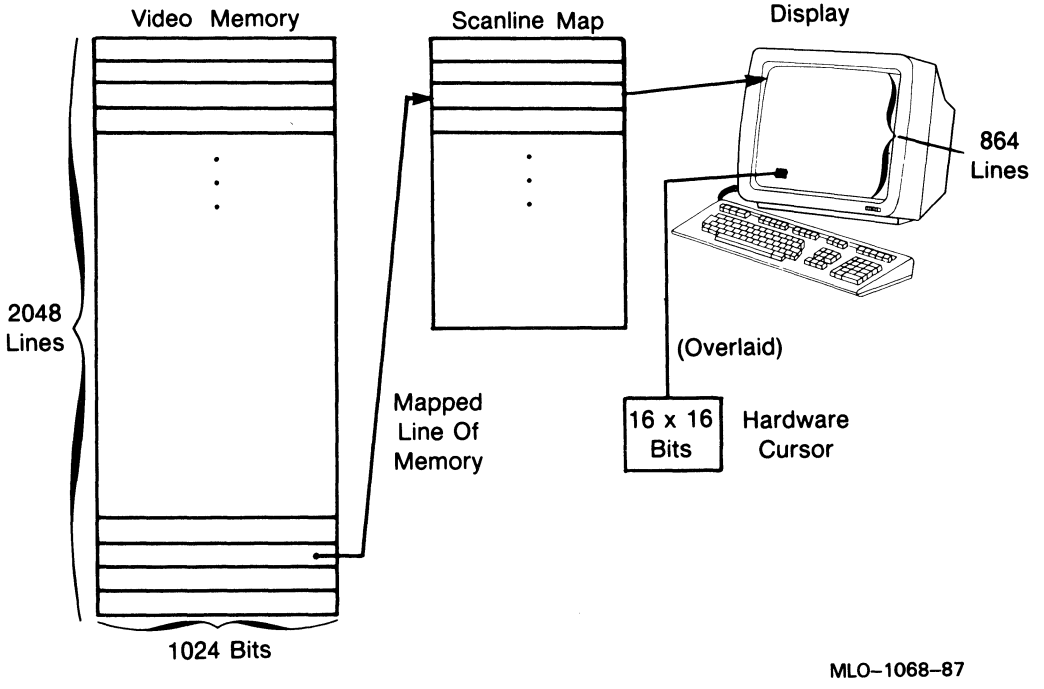
However, since a maximum of 864 lines of video memory can be displayed at any one time and total video memory is 2048 lines, QVSS video memory is referred to in two sections; onscreen and offscreen memory. *Onscreen memory* is any portion of video memory that is currently being displayed. Since the largest VAXstation display is 864 lines, that is the maximum size of onscreen memory.

Offscreen memory is the 1184 lines of video memory that are not displayed. Offscreen memory may be used to store images or fonts that are not currently being displayed. It also plays an important role in the handling of occlusion (see Section 1.3.9).

NOTE: On a VAXstation 2000 monochrome monitor, the system video memory is slightly different. One screen of video memory is available (1024-bit x 864-bit). Also, the hardware scanline map is not available. The system always displays 864 scanlines of video memory with the first scanline starting at the first byte of video memory. Therefore, you cannot change the scanline order.

Figure 1–2 illustrates the layout of QVSS memory.

Figure 1-2 QVSS Video Memory and Scanline Map



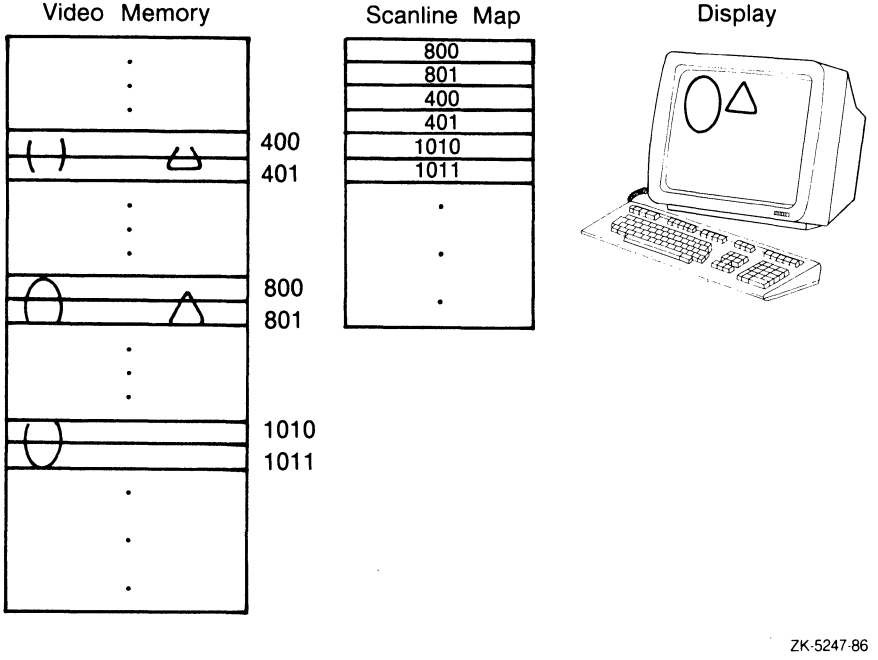
The Hardware Cursor

The cursor that appears on the screen is defined by a separate block of hardware memory. This block is 16 x 16 bits on systems with a single-plane cursor (QVSS) and is 16 x 32 bits on systems with a multiplane cursor (QDSS). This block stores the bitmap image of the cursor pattern. It is *not* part of video memory. The driver uses the hardware to "overlay" the video signal sent to the screen (see Figure 1–2). This arrangement eliminates the need for a save and restore operation in video memory each time the cursor moves or a write to video memory occurs. Section 2.8 describes how to define cursor patterns.

The Scanline Map

Note that the scanline map need not index consecutive lines of video memory. That is, an object may be represented in nonconsecutive lines in video memory (because there is not enough consecutive memory), yet appear on consecutive lines on the screen. Figure 1–3 illustrates how the scanline map properly maps to the screen two objects represented in nonconsecutive memory.

Figure 1-3 Scanline Map Mapping Nonconsecutive Memory



Accessing QVSS Video Memory

QVSS permits direct bitmap access, meaning that an application can directly set bits in the video memory.

An application can write to QVSS video memory using any suitable computer language (FORTRAN, MACRO, and so forth). However, before writing to video memory, an application must issue the Get System Information QIO to obtain the QVSS system block (QVB). The QVB contains all the information an application needs to write to video memory. See Appendix A for a complete description of the QVB.

An application should issue a request for the QVB for each process. The address of the system block does not change. Therefore, a process can obtain the QVB address once, and continue to reference fields in the QVB until the process terminates.

1.3.3.2 How QDSS Uses Video Memory

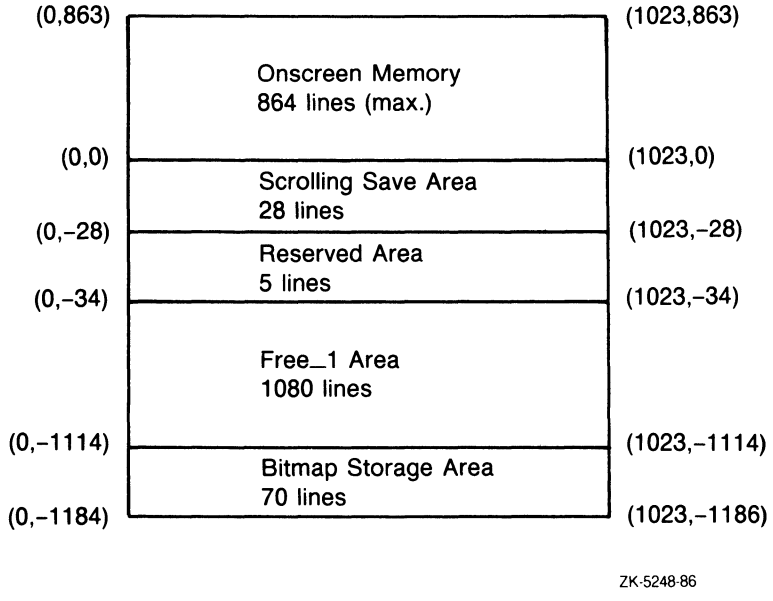
The largest possible VAXstation display is an 1024-pixel by 864-pixel area or 864 lines that are 1024 pixels in length. The full QDSS video memory is a 2048-pixel by 1024-pixel block of memory or, to correspond to the screen, 2048 lines that are 1024 pixels in length. QDSS maps video memory directly to the screen. So, in the case of the largest display, it maps the first 864 lines of video memory to the screen. This portion of video memory is referred to as *onscreen memory.*

The remaining 1184 lines of video memory are referred to as *offscreen memory.* Offscreen memory plays an important role in the handling of occlusion (see Section 1.3.9.) The offscreen portion of the video memory is further divided into the following fixed-length sections:

- Scrolling save area
- Free_1 area
- Bitmap storage area

Figure 1-4 illustrates the layout of QDSS memory and shows relative coordinates for the beginning and end of each area.

Figure 1-4 QDSS Video Map



Note that the lower left-hand corner of onscreen memory is considered to be the coordinate (0,0), corresponding to the lower left-hand corner of the display. (All

viewports are defined relative to this base.) Therefore, any coordinate with a negative Y element is in offscreen memory.

Scroll Area

The driver uses the scroll area to process downward scrolls. This area is reserved for the driver and cannot be accessed by the application.

Free_1 Area

Free_1 is the largest area of free memory. An application can use this memory for any operations it wishes to perform.

Bitmap Storage Area

The driver uses the bitmap area to store bitmaps—fonts, images, and pattern fills. The area consists of a 70-line by 1024-bit block of memory for each plane of memory on the system. Some planes partition these 70-line blocks of memory into two 35-line sections. This permits more fonts to remain in memory. An application uses this area to load any defined bitmaps stored in VAX memory.

You load the information in VAX memory into the bitmap storage area by using the Load Bitmap QIO or the UISDC$LOAD_BITMAP routine.

If a particular bitmap cannot fit into the bitmap storage area, it is the application's responsibility to partition the information into one 70-line section on a single-plane image or two 35-line sections for multiplane images.

Accessing QDSS Video Memory

QDSS does *not* permit direct bitmap access. You draw to video memory using the drawing operation primitive (DOP) interface and you may copy images to video memory using Read Bitmap and Write Bitmap QIOs.

Viewports — Any operation the QDSS driver performs must be directed to the screen by way of a viewport. An application can create viewports or use the default systemwide viewport (the full screen). See Section 1.3.10 for more information about viewports.

Exclusive Bitmap Access — Your application may require exclusive bitmap access to all or part of video memory. When you perform an exclusive-access operation, no other operations should access onscreen memory. Since the QDSS driver controls all access to onscreen memory, an application must always notify the driver of a pending exclusive bitmap operation.

Bitmap Transfers — The QDSS driver can perform three types of bitmap transfers: VAX memory-to-bitmap, bitmap-to-VAX memory, and bitmap-to-bitmap. Transfers are executed using the QIO interface and require exclusive bitmap access for synchronization.

The QDB — An application may issue the Get System Information QIO to obtain the QDSS system block (QDB). The QDB contains information about video memory that an application needs to manipulate video memory. See Appendix B for a complete description of the QDB.

1.3.4 How the Drivers Track Screen Events

The QVSS and QDSS drivers keep track of events that occur on the screen by keeping lists — a list for each type of event. There are four events for which the drivers keep lists:

- Cursor pattern change
- Keyboard entry
- Pointer button transition
- Pointer movement

The drivers also manage a user entry list that your application can use for general storage purposes.

Note that the driver associates each of the first three events listed with a specific region. The keyboard entry event, although not actually associated with a specific region, can be thought of as being associated with the entire screen.

When you define a region with a QIO, the driver uses the information you pass it to construct a *list entry* which it places on the appropriate list. List entries typically contain the following:

- The region definition
- The address of any AST routine defined to take action when the event occurs
- Any AST parameter (token)

When an event occurs the driver searches the appropriate list using the region definition to identify which list entry's AST to issue. For example, if a button transition is detected, the driver searches the button transition list to find if the region that it occurred in has a button transition AST enabled. In the case of the keyboard entry, the first entry on the keyboard list is always invoked.

The following sections provide a brief summary of how the drivers handle each type of event.

1.3.5 Cursor Pattern List

The cursor pattern list is used to determine the pattern of the cursor for a region. You can define the cursor shape and hot spot for a region using the Define Pointer Cursor Pattern QIO. The list entry contains the following:

- Unique channel ID
- The address of a 16 X 16 (16-word) bitmap defining the shape of the cursor
- Cursor hot spot, a point on the 16- by 16-pixel cursor that defines the exact placement of the cursor image
- Cursor background style
- Region definition

See the Define Pointer Cursor Pattern QIO description for more details.

When the driver detects cursor movement, it searches the cursor pattern list for the appropriate ASTs to deliver. If a cursor pattern is to be changed, the bitmap image of the new pattern is located and loaded into the hardware. The new pattern is then superimposed on the appropriate area of the screen.

1.3.6 Keyboard Entry List

The driver uses the keyboard entry list to determine the process to which it should deliver keyboard input. An application can request keyboard ownership by using the Enable Keyboard Input QIO. The driver then delivers input keystrokes to the process by way of AST routines you specify in the QIO. Keyboard list entries are not associated with regions. There is one keyboard associated with each assigned channel.

Each keyboard list entry contains the following:

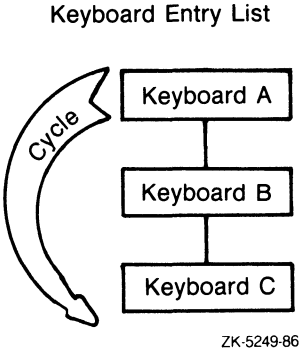
- A unique channel ID
- An AST service routine address
- Address of the longword that receives an input token when the AST routine is called

See the Enable Keyboard Input QIO description for more details.

Unlike other lists, when a keyboard event occurs, the first entry on the list is invoked. The list is not searched. The driver always inserts the active keyboard entry at the beginning of the list.

A keyboard becomes active by popping or cycling operations. Cycling moves the current first entry on the list to the back of the list. Figure 1–5 illustrates cycling.

Figure 1-5 Cycling the Keyboard List



Popping moves an entry from *any* position to the front of the list.

1.3.7 Pointer Button Transition List

The driver uses the pointer button transition list to determine what action to take upon pointer button transitions. A button transition may be either up or down and is detected by the hardware.

A button transition event is triggered when the pointer button is pressed or released within a region defined with the Enable Button Transition QIO. Each pointer button transition list entry contains the following:

- Unique channel ID
- Address of the AST
- Region definition
- Address of the longword to receive the input token indicating which pointer button is activated

When a pointer button transition occurs the driver searches the pointer button transition list for an entry describing the new region. If one is found, the action taken when a pointer button is pressed is determined by the AST routine defined for that region. A token is passed to the specified AST routine to signal which button made a transition, and the type of transition (up or down). See the Enable Button Transition QIO description for details on how button transitions are represented.

If regions for pointer button transitions overlap, priority is given to the first rectangle on the list.

1.3.8 Pointer Movement List

The driver uses the pointer movement list to determine what action to take when the pointing device is moved. You may define a region in which special action can be taken upon pointer movement using the Enable Pointer Movement QIO. Each pointer motion list entry contains

- Unique channel ID
- Address of the AST
- Region definition
- Address of the longword to receive the input token indicating the current physical position of the pointer

The driver searches the list comparing the current pixel position of the pointer against the region definitions of all list entries. The current location of the pointer determines what event to trigger. If regions for pointer motion events overlap, priority is given to the first region on the list.

When the pointer cursor moves outside a currently active region, the AST is passed a special exit token of –1, to notify the process that the cursor has left the region. A process may wish to perform some cleanup functions when the pointer leaves the region.

1.3.9 Occlusion

In a windowing system, it is possible for more than one window to address the same area of the screen. However, the drawing of only one window can be displayed in a particular area at any given time. The case when one window display occludes, or hides, a part or all of another display is called occlusion.

You must write your application to handle occlusion (unless it has only one window or it can guarantee that windows never overlap). The QDSS driver offers some software support for handling occlusion. See Section 1.3.10.2. The QVSS driver, because it is a direct bitmap access system, does not offer any support for handling occlusion; that is solely the responsibility of the application.

1.3.10 What Is a Viewport?

This section is QDSS-specific.

The QDSS driver directs various operations to the screen by way of *viewports*. A viewport is a rectangular area defined within video memory. Viewports may be user-defined or *systemwide* viewports.

A user-defined viewport can be specified anywhere in video memory. The dimensions of the viewport are defined by using the Set Viewport Region QIO. Logically, a viewport is a single rectangular region. However, in order to handle occlusion, you can use Set Viewport Region to define a viewport as one or a number of distinct *update regions* (see the next section). These regions together describe the size of the viewport and its relative location in video memory.

The systemwide viewport is one update region that encompasses all available video memory and can be considered the system default viewport. If no other viewport is defined, all drawing operations are directed to the systemwide viewport.

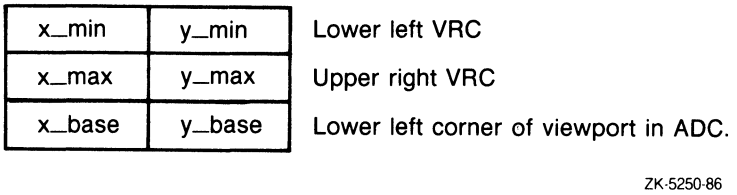
The coordinates of all drawing operations directed to a particular viewport are specified relative to the lower left-hand corner, or base, of the viewport. Such operations are said to be viewport relative.

1.3.10.1 Viewport Update Regions

As described above, update regions store all the coordinate information necessary to define a viewport. An application uses the data structure illustrated in Figure 1–6 to define a viewport update region. The driver uses this information when mapping the viewport-relative coordinates (VRC) of the viewport to the absolute device coordinates (ADC) of the display.

The coordinate information stored in an update region contains the location of the viewport in video memory (x_base, y_base) and the extents of the viewport (x_min, y_min, x_max, and y_max).

Figure 1-6 Viewport Update Region Data Structure



1.3.10.2 Using Update Regions for Occlusion

You can define viewports that overlap the same area of onscreen memory. When this occurs, the portion of the viewport that is not visible is considered "occluded" and is hidden behind the other viewport on the screen. Since only one viewport can occupy an area of memory at any given time, you cannot write to the onscreen location of the overlapped portion of the occluded viewport.

An application may direct writing operations to the occluded portion of the screen even though it cannot display them while the region is occluded. To keep track of any drawing operations that may be directed to an occluded portion of a viewport, your application must divide the occluded viewport into several update regions. The accessible regions are updated immediately in onscreen memory, and the operations to an occluded region are either performed in offscreen memory or deferred until memory is available. Chapter 2 describes how to handle occlusion.

1.3.11 QDSS Drawing Operation Queues

As stated in a previous section of this chapter, the QDSS driver provides a request queue interface that executes drawing operation primitives (DOPs). Chapter 5 describes how to construct DOPs and enter them on the request queue. This section describes the mechanisms used for executing DOPs.

There are three queues that are relevant to the execution of DOPs:

- The request queue
- The return queue (actually two queues)
- The deferred queue

The following sections describe each queue.

1.3.11.1 The Request Queue

A request queue is a doubly linked list of DOPs that have been submitted for execution. Each defined viewport (including the systemwide viewport) has a request queue associated with it. Any drawing operation to be executed within a viewport must be entered on that viewport's request queue. An application associates a viewport with a request queue by getting a viewport ID for the viewport. (This is done using the Get Viewport ID QIO.) The viewport ID is actually the address of the DOP queues data structure which contains the starting address of the request queue (as well as that of the return queues). See Appendix B for a full description of this data structure.

You may enter a DOP on the request queue in any of the following ways:

- Using the UISDC interface (see Chapter 5)
- Issuing the Insert DOP QIO

- Using the MACRO instruction INSQUE

At specific intervals, the driver scans all request queues checking for work. If any queue contains DOPs, the driver removes the DOPs from the queue and performs the specified operations. The order in which the packets are stored on the queue is the order in which the drawing operations occur.

Certain screen management circumstances require that the processing of request queues be stopped. Sometimes it is appropriate to stop a single viewport's request queue, sometimes all viewport request queues must be stopped. The QIO interface provides a number of QIOs that manipulate the request queue. (A small number of DOPs also manipulate the request queue — in a limited way.) Chapter 2 describes the circumstances under which an application manipulates the request queue.

1.3.11.2 The Return Queue

A DOP is a data structure and must have storage allocated for it. If you were to process a large number of DOPs without any restrictions, they could consume all available system memory (or enough to cause considerable performance degradation).

The QDSS driver provides the return queue as a way for an application to reuse storage allocated for DOPs. The return queue, like the request queue, is a doubly linked list. Once a DOP is completely processed, the driver removes it from the request queue and inserts it on the return queue (by simply updating the links). A DOP on the return queue is referred to as a *free DOP*.

An application can save storage by checking the return queue for used DOP storage before allocating memory for new drawing operations.

Allocating DOPs

You allocate storage for DOPs using either the UISDC interface (see Chapter 5) or using memory allocation system routines. When you allocate a DOP, you allocate one of two fixed sizes, small or large. You define the sizes of a large and a small DOP by initializing the DOP size fields of the DOP queue structure and the request queue structure. This structure is described in Appendix B.

All applications should first check the return queue for reusable DOP storage before allocating new storage. (The UISDC interface does this.) An application can force the driver to wait for a DOP from the return queue by using the Get Free DOPs QIO. One of the parameters for this QIO allows an application to specify the number of DOPs to wait for on the return queue before returning control to the application. The application can then reuse the storage by removing a DOP from the return queue.

This feature can prevent a situation where too much system memory is allocated. For example, assume an application allocates 300 DOPs for processing on the request queue and before processing is complete needs more DOPs for additional operations. If the application were allowed to allocate all the additional DOPs it requires the application might consume all available system memory. However, specifying a high

number of DOPs to the Get Free DOPs QIO forces the application to halt further memory allocation until that number of DOPs is available for reuse.

Actually Two Queues

As mentioned above, DOPs are allocated in two sizes. It is only logical then that the return queue is actually two queues, one to which small DOPs are returned, and another to which large DOPs are returned. When you issue the Get Free DOPs QIO, you specify the number of DOPs you wish to wait for as well as the specific return queue.

Alternate return queue

By default, a return queue is associated with each viewport through the DOP queue structure. However, you may use an alternate return queue if you wish to share return queues on a per-process or systemwide basis. To do so, you specify the address of a return queue structure as the fourth parameter of the Get Viewport ID QIO when you create the viewport. The return queue section of the viewport's DOP queue structure is ignored in this case. See Appendix B for descriptions of these structures.

1.3.12 The Deferred Queue

When a portion of a viewport is occluded onscreen, the first option for writing to it is to write to an update region in offscreen memory. However, sometimes offscreen memory is so crowded that you cannot keep an update region in offscreen memory. In this case, you must save the writing operations on the *deferred queue.*

The driver stores all operations directed to a portion of a viewport that is not currently available to the QDSS hardware on the deferred queue. Deferred queue operations can be executed at any time the previously occluded portion of the viewport becomes available.

To prevent the deferred queue from consuming system resources, an application should update occluded viewports when the queue is full. The Notify Deferred Queue Full QIO notifies an application the deferred queue is full. The Execute Deferred Queue QIO executes the operations on the deferred queue. For additional information on deferred queue operations see Chapter 2 of this manual.

Chapter 2

How to Program to the Driver

This chapter describes how to perform programming tasks that are commonly used when writing to the video device drivers. Tasks that are common to both drivers are described first. Tasks that are specific to either the QVSS or QDSS driver are clearly noted.

This chapter details how specific QIOs and DOPs are used in combination to perform tasks. No attempt is made in this chapter to fully describe these QIOs and DOPs because Chapter 3 and Chapter 4 describe each function of the QIO interface in detail. Chapter 5 describes the DOPs that are available (on a QDSS system) for drawing to the display.

2.1 Initializing the Screen

Before an application can write to the screen, it must first initialize the screen. Initialization places the VAXstation screen in a known state. Once the initialization is complete, the application can issue additional QIOs and begin screen operations. Failure to initialize the screen before doing a draw operation will result in undefined behavior for the drawing operation.

To initialize the screen, use the Initialize QIO. This QIO has no parameters and is invoked only once for each application.

2.2 Accessing the System Information Block

The system information block (the QVB or QDB, depending on which type of system you have) is a data structure that both drivers use to store information describing the current state of video memory. This data structure consists of a number of fields, each of which is associated with a symbolic constant that is used to reference the field. See Appendixes A and B for a full illustration and explanation of each field in these data structures.

To get the address of the system information block, use the Get System Information QIO. This QIO returns a descriptor that contains the address and size of the block. To access any field in the block, use the returned address and the symbolic constants defined for that field.

How to Use the QDB

A QDSS application may or may not need to access the QDB. If it is only using the UISDC DOP interface there is no need. However, under some circumstances, for example, if it manipulates the pointer position or requires tablet information, it may need system information stored in the block. Example 2–5 uses the QDB to get the systemwide viewport ID.

How to Use the QVB

Typically, before a QVSS application can perform a drawing operation, it must obtain specific video memory information from the QVB — the starting address of video memory and the address of the scanline map. A QVSS application needs the starting address of video memory in order to set bits (draw) in memory. It needs the address of the scanline map to map to the screen any lines in memory it wishes to display. How to use the QVB for drawing operations is described in Section 2.10. The following code segment shows a call that obtains the descriptor for the QVB.

```
! Declare QDB descriptor and buffer
INTEGER*4 QDB_DESC(2)
.
.
.
! Obtain a channel for the call
CALL SYS$ASSIGN ('SYS$WORKSTATION',     ! device name
2                CHAN,,)                ! channel
```

```
! Get the QVB
CODE = IO$_SENSEMODE
STATUS = SYS$QIOW (,
2                   %VAL(CHAN),            ! channel
2                   %VAL(CODE),            ! QIO function code
2                   ,,,
2                   %VAL(IO$C_QV_GETSYS),
2                   QVB_DESC,,,,)          ! address of descriptor
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Extract from the QVB
CALL EXTRACT_QVB (%VAL(QVB_DESC(2)),     ! pass address by value
2                 VIDEO_ADDR,            ! video memory buffer
2                 SCANLINE_ADDR)         ! scanline map buffer
.
.
.
! *************************
! * EXTRACT_QVB SUBROUTINE *
! *************************
FUNCTION EXTRACT_QVB (QVB, VIDEO, SCAN)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ QVB
RECORD /QVB_COMMON_STRUCTURE/ QVB

VIDEO = QVB.QVB$L_MAIN_VIDEOADDR
SCAN = QVB.QVB$L_MAIN_MAPADDR

RETURN
END
```

2.3 Using Channels with the Video Device Drivers

Channel numbers are used by the drivers to identify application list entries. Recall the way that the drivers use entry lists; they keep track of different events by putting entries for the events on lists (Section 1.3.4). When an application places an entry on a list (using the QIO interface), it associates the entry with a channel number (using the *chan* parameter of the QIO). To obtain a unique channel number, use the $ASSIGN system service.

The channel number associated with each entry an application puts on a list must be unique to that list. That is, if a process creates an entry on the keyboard list using channel 1, it cannot create another entry on the keyboard list using channel 1. It must obtain another channel to create another entry. The driver uses the channel number to uniquely identify the entries when adding or deleting entries.

However, the same channel number can be used across lists. For example, an application can create an entry on the button transition list, the keyboard entry list, and the cursor pattern list, each using channel 1.

2.4 Using the Keyboard

Driver keyboard functions permit an application to:

- Receive keyboard input
- Modify the keyboard characteristics of a keyboard
- Modify keyboard character sets
- Compose characters that do not exist as standard keys

The following sections describe these capabilities.

2.4.1 Receiving Keyboard Input

To receive input from a keyboard an application must explicitly enable the keyboard for input using the Enable Keyboard Input QIO. Enabling a keyboard creates an entry on the keyboard entry list. When you enable a keyboard you may specify:

- The address of a four longword AST specification block. The four longwords contain

  1. The address of an AST routine that determines what action to take when input is received. This type of AST is referred to as a keystroke AST because it is fired for each keystroke that is input.

     If no AST routine is specified, input will be stored in the typeahead buffer, and delivered either when an AST region is declared using the same channel or when a Get Next Input Token QIO is issued.

  2. An optional user-defined AST parameter that is delivered to the AST.
  3. Access mode at which to deliver the AST (maximized with the current access mode).
  4. A zero.

- The address of an AST routine that is invoked whenever the entry is made active (brought to the top of the list). This type of AST is referred to as a request AST (or control AST). It is fired only once for each entry activation. Only one keyboard can be active at any one time. Typically, the request AST performs any necessary actions that the enabled keyboard requires, for example, popping an obscured window so that displayed input may be seen.
- Address of a keyboard characteristics block that describes the characteristics of the keyboard. Each keyboard has a set of characteristics associated with it (key click, auto repeat, and so on) that are defined in this data structure. You may choose to take the default characteristics or modify the defaults. Section 2.4.2 discusses keyboard characteristics in detail.

By default, an entry is placed at the top of the list when a keyboard is enabled. However, an application may determine the position of the entry in the list by specifying optional modifiers to the QIO. The modifiers can be used to:

- Cycle the list — top entry is moved to the end of the list (QV_CYCLE)
- Place the entry last on the list (QV_LAST)
- Delete the entry (QV_DELETE)
- Purge the typeahead buffer (QV_PURG_TAH)

Typically, an application defines one keyboard for each window. However, it may define the characteristics of each keyboard differently from window to window. This feature permits you to create "virtual" keyboards. Although there is one physical keyboard, there may be a number of different keyboards defined. An application may enable any number of keyboards — as long as it keeps track of them. When a keyboard is no longer needed, you delete it either by using the QV_DELETE modifier with the QIO or by deassigning the associated channel.

The following example shows a typical assignment of two terminal channels, keyboard requests on those channels, and the associated AST routines. (Appendix D of this manual shows this example in the context of a complete applications program.)

Example 2-1 Enabling Keyboard Requests

```
    .
    .
    .
P2_BLOCK2:                              ; AST specification block 2
        .LONG   KBD_AST                 ; AST address
        .LONG   ACK2                    ; AST parameter
        .LONG   0                       ; AST delivery mode
        .LONG   CHARACTER               ; input token
ACK2:   .ASCID  /INPUT ACKNOWLEDGED CHANNEL 2/  ;AST parameter message

P3_BLOCK:                               ; control AST specification
                                        ; block - for both ASTs
        .LONG   CTL_AST                 ; control AST address
        .LONG   0                       ; AST parameter
        .LONG   0                       ; AST delivery mode
        .LONG   0                       ; must be zero
    .
    .
    .
SET_KBDAST:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; assign channel using
                   CHAN=KBD_CHAN2       ; logical name and
                                        ; channel number
        BLBS    R0,20$                  ; no error if set
        BRW     ERROR                   ; error
    .
    .
    .
20$:    MOVL    #IO$C_QV_ENAKB,R0       ; enable keyboard AST
                                        ; request to R0
        $QIOW_S CHAN=KBD_CHAN2,-        ; assigned channel
                FUNC=#IO$_SETMODE,-     ; set mode QIO
                P1=(R0),-               ; keyboard AST request
                P2=#P2_BLOCK2,-         ; user AST routine
                P3=#P3_BLOCK            ; control AST routine
        BLBS    R0,30$                  ; no error if set
        BRW     ERROR
    .
    .
    .
```

(Continued on next page)

Example 2-1 (Cont.) Enabling Keyboard Requests

```
KBD_AST:
F5_AST:
        .WORD
        PUSHL   4(AP)                   ; send acknowledgment
        CALLS   #1,G^LIB$PUT_LINE       ; message
        BLBS    R0, 10$
5$:     BRW     ERROR

10$:    CMPW    #KEY$C_F5,CHARACTER     ; was F5 typed?
        BNEQ    20$
        BSBW    CYCLE_KBD               ; cycle the keyboard list
        BRB     40$                     ; and exit

20$:    PUSHAL  DESC                    ; send character typed
        CALLS   #1,G^LIB$PUT_LINE
        BLBC    R0,5$

        CMPB    #^A/C/,CHARACTER        ; was a "C" typed?
        BNEQ    30$
        BSBW    CYCLE_KBD               ; cycle the keyboard list
        BRB     40$

30$:    CMPB    #^A/F/,CHARACTER        ; was an "F" typed?
        BNEQ    40$
        $SETEF_S EFN=#2                 ; yes, exit program

40$:    RET
    .
    .
    .
CTL_AST:
        .WORD
        PUSHAL   CYCLE                  ; send acknowledgment
        CALLS    #1,G^LIB$PUT_LINE      ; message
        BLBS     R0, 10$
5$:     BRW      ERROR

10$:    RET
    .
    .
    .
```

2.4.2 Keyboard Characteristics

By default, keyboards have a set of characteristics associated with them. The default characteristics of a keyboard are defined by a data structure referred to as the system characteristics block. Appendix A illustrates this data structure and lists the keyboard characteristics and their defaults.

An application may modify the default characteristics by specifying a system characteristics block as the fourth parameter of the Modify Systemwide Characteristics QIO. This block consists of four longwords:

- The first longword contains a bit mask of the characteristics to enable.
- The second longword contains a bit mask of the characteristics to disable.
- The third longword is the key-click volume value in the range 1 to 8 (1 being loudest).
- The fourth longword is the screen saver time, in minutes.

To enable or disable default values, specify the predefined QVBDEF constant associated with each characteristic (also listed in the appendix) in the proper longword. If the key-click or screen saver characteristics are enabled, their respective values in the third and fourth longword are used.

Once you have modified the systemwide defaults, any keyboard enabled without specifying keyboard characteristics takes the new default values.

An application can also define the keyboard characteristics (auto repeat, key-click sound, function key transition) for a particular keyboard entry by specifying the address of a keyboard characteristics block as the fourth parameter of the Enable Keyboard Input QIO. This block also consists of four longwords:

- The first longword contains a bit mask of the characteristics to enable.
- The second longword contains a bit mask of the characteristics to disable.
- The third longword is the key-click volume value in the range 1 to 8 (1 being loudest).
- *The fourth longword must be zero.*

The characteristics you can enable for a specific keyboard are the same as those for the systemwide defaults, except for the screen saver time — it is a systemwide characteristic (there is only one screen). See Appendix A.

These characteristics are associated with the enabled keyboard. Whenever the keyboard is active, the physical keyboard will reflect these characteristics. To clarify, if an application enables two keyboards, one with autorepeat enabled and the other with autorepeat disabled; when the first keyboard is active any key held down will be sent repeatedly; when the other keyboard is active, the key will only be input once.

To modify the characteristics of an existing keyboard, use the Modify Keyboard Characteristics QIO, specifying a keyboard characteristics block as the fourth parameter. Note that this QIO may also be used to change the keystroke AST and request AST associated with a keyboard.

2.4.3 Modifying the Keyboard Table

The keys on the main keypad array of the physical keyboard are programmable. That is, an application may define the keys of the keyboard so that they are associated with any of the 255 characters in the multinational character set (including diacritical characters). (Appendix G contains a table showing the multinational character set.) An application may define several character sets to be accessed at different times by the same physical keyboard.

You define a character set by constructing a data structure referred to as a keyboard table and then loading the new table using the Load Keyboard Table QIO.

2.4.3.1 Constructing a Keyboard Table Using Macros

You construct a keyboard table by initializing it with the default table values and then overriding any values you wish to modify. VWS provides macros to generate keyboard tables in SYS$LIBRARY:$QVBDEF.MLB. They are:

- VC$KEYINIT — Initializes the table
- VC$KEY — Loads individual key definitions
- VC$KEYEND — Terminates the table

These macros also appear in Appendix E.

Initializing a Table

To initialize a table, call VC$KEYINIT. This macro has one parameter, the address of the table, which it *returns* after allocating space and initializing the table. It is recommended that you specify this parameter and use the returned address when loading the table. By default, the system loads the North American keyboard table.

Loading Key Definitions

To load new key definitions, call VC$KEY. This macro has several parameters that permit you to define the various states of a given key. Note that you only need to modify the keys that are different from the default keys. Loading key definitions *overrides* the default definitions loaded by VC$KEYINIT.

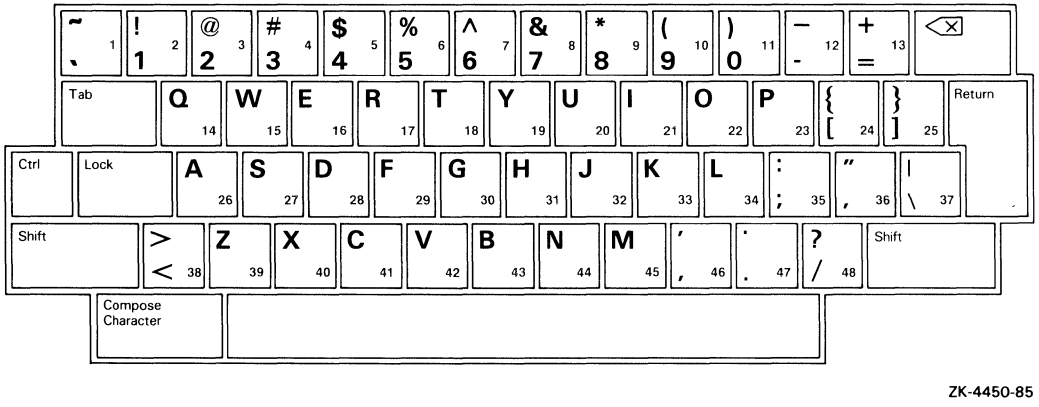
A key on the keyboard can have eight different states, depending on how the Shift, Control, and Lock keys are pressed in combination with the key. For each key, the keyboard table associates each state with a one-byte ASCII value representing a character from the multinational character set, so each key is described by a quadword in the table. The following table lists the states a key can have along with the byte within the quadword that describes each state.

Byte	State
1	Value of key
2	Value of key if Shift key is also pressed
3	Value of key if Control key is also pressed
4	Value of key if both Shift and Control keys are also pressed
5	Value of key if Lock is also pressed
6	Value of key if both Lock and Shift keys are also pressed
7	Value of key if both Lock and Control keys are also pressed
8	Value of key if Lock, Shift, and Control keys are also pressed

When you load a key using VC$KEY, you specify nine parameters, the ordinal key position, and the ASCII value associated with each of the eight states.

Figure 2-1 shows the order of keys in the keyboard table. It illustrates the relationship of the physical key on the keyboard to the ordinal key position in the keyboard table (numerals in red). This particular table corresponds to the North American layout of characters on the keyboard.

Figure 2-1 Keyboard Table Layout



ZK-4450-85

The following is an example of loading the tenth key of the keyboard table:

```
VC$KEY  10,-        ; ordinal key position
        <^a/9/>,-   ; key = 9
        <^a/)/>,-   ; Shift/key = )
        <^x0FF>,-   ; Control/key = undefined
        <^x0FF>,-   ; Shift/Control/key = undefined
        <^a/9/>,-   ; Lock/key = 9
        <^a/)/>,-   ; Lock/Shift/key = )
        <^x0FF>,-   ; Lock/Control/key = undefined
        <^x0FF>     ; Lock/Shift/Control/key = undefined
```

Note that the hexadecimal value 0FF is used to denote an undefined key (meaning no character is delivered).

The following table shows the decimal values that should be used to represent diacritical characters. (Section 2.4.4 contains additional information on diacritical characters.)

Diacritical Mark	Equivalent Character	Decimal Value
Diaeresis (umlaut)	Ä	128
Acute accent	´	129
Grave accent	`	130
Circumflex accent	^	131
Tilde	~	132
Ring	°	133
(Reserved)		134–159

Terminating a Table

To terminate a keyboard table, call VC$KEYEND. This macro has one parameter, the length of the table, which it *returns*. It is recommended that you specify this parameter and use the returned length when loading the table.

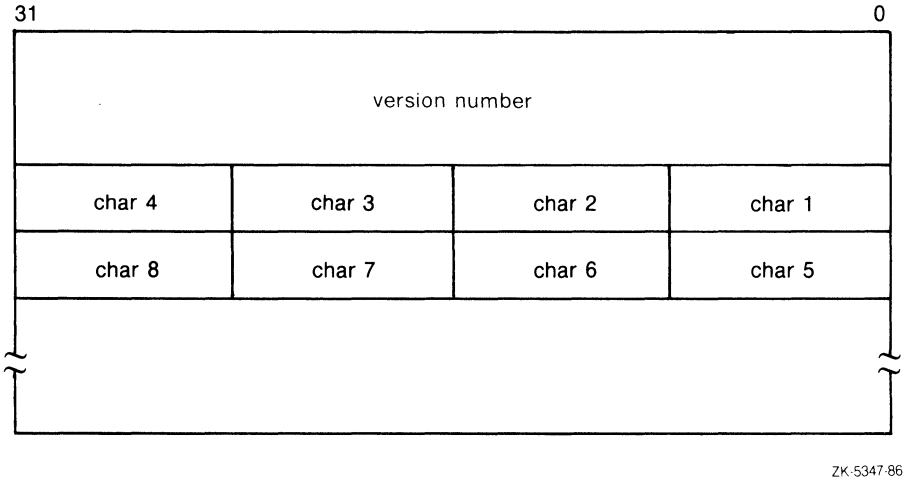
The following example segment shows how the North American keyboard layout can be modified. (Appendix D of this manual shows this example in the context of a complete applications program.)

```
    .
    .
    .
VC$KEYINIT   KB_LAYOUT_TBL    ; generate the new table
                              ; modify only the characters
                              ; specified
    .
    .
    .
VC$KEY  10,<^a/9/>,<^a/)/>,<^x0FF>,<^x0FF>,-
           <^a/9/>,<^a/)/>,<^x0FF>,<^x0FF>
VC$KEY  11,<^a/0/>,<^a/=/>,<^x0FF>,<^x0FF>,-
           <^a/0/>,<^a/=/>,<^x0FF>,<^x0FF>
VC$KEY  12,<^x081>,<^a/?/>,<^x0FF>,<^x0FF>,- ;diacritical (')
           <^x081>,<^a/?/>,<^x0FF>,<^x0FF>
VC$KEY  13,<^x082>,<^x083>,<^x01E>,<^x01E>,- ;diacriticals (` ^)
           <^X082>,<^x083>,<^x0FF>,<^x0FF>
VC$KEY  19,<^a/z/>,<^a/Z/>,<^x01A>,<^x01A>,-
           <^a/z/>,<^a/Z/>,<^x01A>,<^x01A>
VC$KEY  24,<^x0E8>,<^x0FC>,<^x0FF>,<^x0FF>,-
           <^x0E8>,<^x0FC>,<^x0FF>,<^x0FF>
VC$KEY  25,<^x080>,<^x084>,<^x0FF>,<^x0FF>,- ;diacriticals (" ~)
           <^x080>,<^x084>,<^x0FF>,<^x0FF>
    .
    .
    .
VC$KEYEND    KB_LAYOUT_TBL_LEN        ; end the table,
                                      ; and determine its length
```

2.4.3.2 Constructing a Keyboard Table Without Macros

If you construct a keyboard table without using the provided macros, it must conform to the structure shown in Figure 2–2.

Figure 2–2 Keyboard Table Description



- The first quadword of the table must contain the version number for the table. Typically this value will be 1.
- Each subsequent quadword describes the eight states of a key in the main array of the keyboard, in the order shown in Figure 2–1.
- Every key must be defined.

2.4.3.3 Loading a Keyboard Table

To load a keyboard table, use the Load Keyboard Table QIO, specifying the address and size of the table and the channel of the keyboard entry with which you wish to associate the table. Note that the address and length are returned by the VC$KEYINIT and VC$KEYEND macros, respectively.

Once a keyboard table is loaded and the associated keyboard becomes active, the physical keyboard reflects the definitions in the table.

You can also revert to the default keyboard table by calling the Revert to Default Keyboard Table QIO. This QIO also returns the space used by a private table (if one was loaded) to pool.

The following example shows a typical routine for loading a keyboard table. (Appendix D of this manual shows this example in the context of a complete applications program.)

```
    .
    .
    .
SET_FRENCH_KB:
        MOVL    #<IO$C_QV_LOAD_KEY_TABLE>, R0
        $QIOW_S CHAN = KBD_CHAN1, -         ; change the keyboard
                FUNC = #IO$_SETMODE, -      ; layout
                P1 = (R0), -
                P2 = #KB_LAYOUT_TBL_LEN, - ; keyboard table size
                P3 = #KB_LAYOUT_TBL         ; keyboard table
                                            ; address
        BLBS    R0,5$                       ; no error if set
        BRW     ERROR
5$:     RSB
    .
    .
    .
```

2.4.4 Composing Nonstandard Characters

Compose sequences permit you to define combinations of keys to represent multinational characters not already defined as standard keys in the keyboard table.

There are two types of compose sequences: two-stroke sequences and three-stroke sequences. Three-stroke sequences can be used on all keyboards. To perform a three-stroke sequence, you press the Compose key and then press two standard keys. Two-stroke sequences can be used on all keyboards except the North American keyboard. To perform a two-stroke sequence, you press a *diacritical* mark and then press a standard key.

A diacritical mark is one of the following nonspacing characters:

grave accent
acute accent
circumflex accent
tilde
diaeresis (umlaut)
ring

Diacritical marks are available on all but the North American keyboard. (This is why two-stroke sequences are not possible on the North American keyboard.) The diacritical marks vary among the keyboards according to the relative usage of characters with diacritical marks. Users should note that only one of several characters shown on a key cap may be a diacritical mark; some keyboards have keys that contain both a standard character and a diacritical mark.

You define compose sequences by constructing a data structure referred to as a compose sequence table (either two-stroke, three-stroke, or both) and then loading the table with the Load Compose Sequence Table QIO.

2.4.5 Constructing Compose Sequence Tables

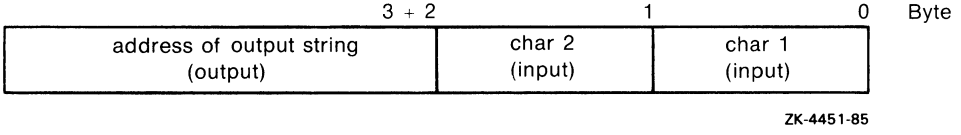
A *compose sequence table* lists a series of compose sequences. The structures of a three-stroke table and a two-stroke table differ slightly.

Three-Stroke Compose Sequence Table Structure

Three-stroke compose tables have three parts:

1. The first part of the table is a longword containing the version number for the table. (Typically this value will be 1.)
2. The second part of the table is a series of longwords, each of which describes a compose sequence. These longwords list the two standard keys used in the compose sequence and hold an address that points to the associated output string. They have the format shown in Figure 2–3.

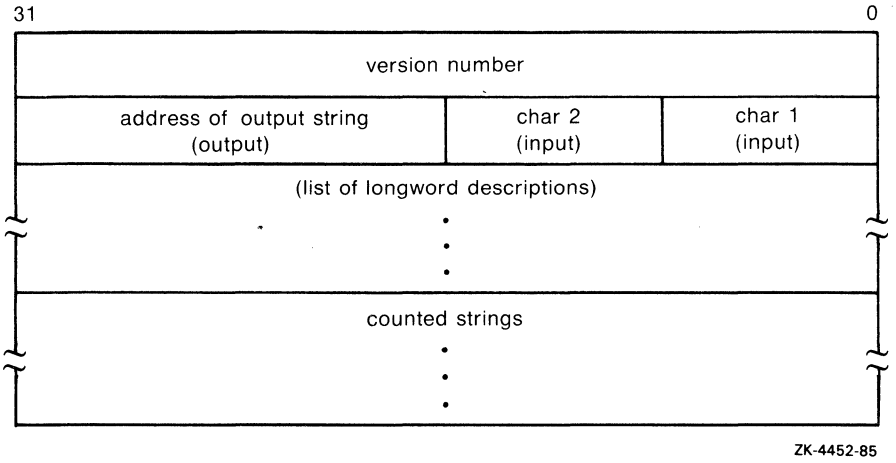
Figure 2–3 Three-Stroke Compose Sequence Table Description



3. The third part of the table is the counted string that bytes 2 and 3 of the longword (address of output string) point to. All the counted strings must be grouped together, and must follow the list of longwords that describe the compose sequences.

Figure 2–4 shows the structure of an entire three-stroke compose sequence table.

Figure 2-4 Three-Stroke Compose Sequence Table



Two-Stroke Compose Sequence Table Structure

Two-stroke compose sequence tables have four parts:

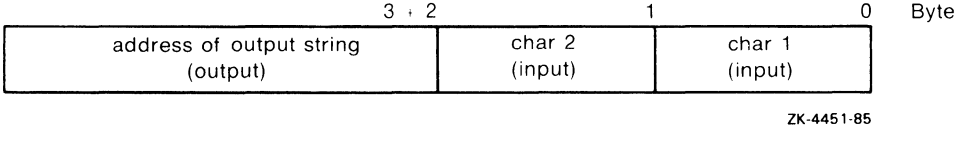
1. The first part of the table is a longword containing the version number for the table. (Typically this value will be 1.)
2. The second part of the table is a 32-byte diacritical table. This table defines which characters are diacriticals. Each bit in the diacritical table corresponds to the equivalent ASCII character code in the multinational character set. If a bit is set in this table, the corresponding character is considered a diacritical character. Note that this means it is possible to define nonstandard diacritical characters. For example, if bit 65(decimal) were set, then the uppercase letter "A" would be a diacritical character. If bit 112(decimal) were set, then the lowercase letter "p" would be a diacritical character.

   In order to support *standard* diacritical characters, represent the characters using the decimal values shown in the following table.

Diacritical Mark	Equivalent Character	Decimal Value
Diaeresis (umlaut)	Ä	128
Acute accent	´	129
Grave accent	`	130
Circumflex accent	^	131
Tilde	~	132
Ring	°	133
(Reserved)		134–159

3. The third part of the table is a series of longwords, each of which describes a compose sequence. These longwords list the diacritical key and the standard key used in the compose sequence and hold an address that points to the associated output string. They have the format shown in Figure 2–5.

Figure 2–5 Two-Stroke Compose Sequence Table Description



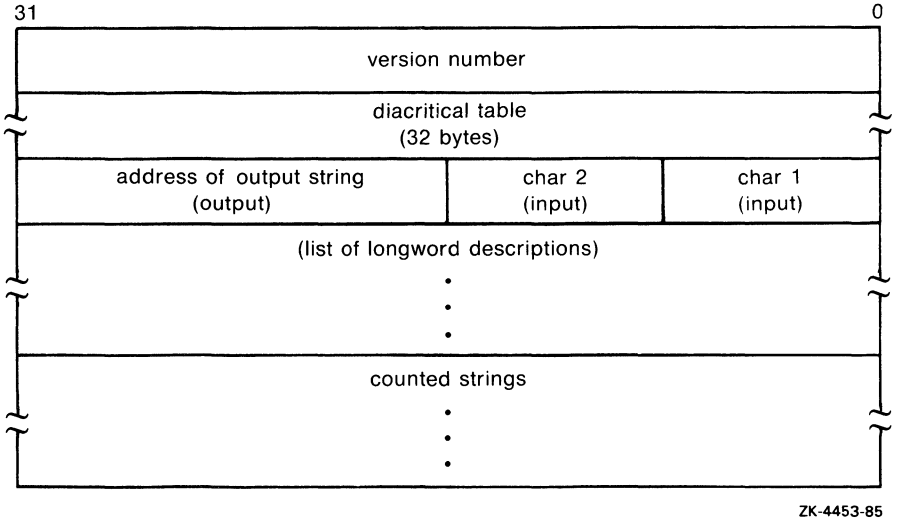
The list of longwords in the table must be in ascending order by ASCII collating sequence using both input characters, as shown in the following example.

Incorrect Table	Correct Table
s s 1 ß	A E 1 Æ
A E 1 Æ	A ~ 1 Ã
^ a 1 â	^ a 1 â
A ~ 1 Ã	s s 1 ß

4. The fourth part of the table is the series of counted strings that bytes 2 and 3 of each longword (address of output string) point to. All the counted strings must be grouped together, and must follow the list of longwords that describe the compose sequences.

Figure 2–6 shows the structure of an entire two-stroke compose sequence table.

Figure 2-6 Two-Stroke Compose Sequence Table



2.4.5.1 Constructing Compose Sequence Tables Using Macros

You construct a compose table by initializing it and then loading the sequences you wish to define. VWS provides macros to generate compose tables in SYS$LIBRARY:$VWSSYSDEF.MLB. They are

- VC$COMPOSE_KEYINIT — Initializes the table
- VC$COMPOSE_KEY — Loads individual sequence definitions
- VC$COMPOSE_KEYEND — Terminates the table

These macros also appear in Appendix F.

Initializing a Table

To initialize a table, call VC$COMPOSE_KEYINIT. This macro has two parameters:

- The address of the table, which it *returns* after allocating space and initializing the table. It is recommended that you specify this parameter and use the returned address when loading the table.
- The Compose_2 flag, which, if set equal to YES indicates that a two-stroke sequence table should be built. If the flag is not set, a three-stroke table is built.

Loading a Compose Sequence

To load a compose sequence, call VC$COMPOSE_KEY. This macro has four parameters that permit you to define the two input characters (either the two standard keys for a three-stroke sequence or the diacritical and standard key for a two-stroke sequence), the output string, and the output string length. The following is an example of loading a three-stroke compose sequence:

```
VC$COMPOSE_KEY <^a/A/>,- ; input A
               <^a/"/>,- ; input "
                      ,- ; default output length
               <^xc4>    ; output character
```

Terminating a Table

To terminate a keyboard table, call VC$COMPOSE_KEYEND. This macro has one parameter, the length of the table, which it *returns*. Typically, you specify this parameter and use the returned length when loading the table.

2.4.5.2 Loading a Compose Table

To load a compose table, use the Load Compose Table QIO, specifying the address and size of the table and the channel of the keyboard entry with which you wish to associate the table. Note that the address and length are returned by the VC$COMPOSE_KEYINIT and VC$COMPOSE_KEYEND macros, respectively.

The MicroVMS Workstation is shipped with copies of the DIGITAL standard three-stroke and two-stroke compose tables residing within the driver. These are the default tables until alternate default tables are loaded. Note that DIGITAL standard two-stroke compose sequences are not supported on the North American keyboard.

You can also revert to the default compose table by calling the Revert to Default Compose Table QIO.

The following example shows how to load a three-stroke compose table. (Appendix D of this manual shows this example in the context of a complete applications program.)

```
    .
    .
    .
SET_COMPOSE3_TABLE:
        MOVL    #<IO$C_QV_LOAD_COMPOSE_TABLE>, R0
        $QIOW_S CHAN = KBD_CHAN1, -         ; change the compose table
                FUNC = #IO$_SETMODE, -
                P1 = (R0), -
                P4 = #COMPOSE3_TBL_LEN, - ; three-stroke table size
                P5 = #COMPOSE3_TBL          ; three-stroke table addr
        BLBS    R0,5$                       ; not set on error
        BRW     ERROR
5$:     RSB

        VC$COMPOSE_KEYINIT  COMPOSE3_TBL  ; generate an
                                          ; empty table
                                          ; fill the table here
                                          ;
        VC$COMPOSE_KEY  <^a/A/>,<^a/"/>,,<^xc4>
        VC$COMPOSE_KEY  <^a/A/>,<^a/'/>,,<^xc1>
        VC$COMPOSE_KEY  <^a/A/>,<^a/*/>,,<^xc5>
        VC$COMPOSE_KEY  <^a/A/>,<^a/A/>,<@>
        VC$COMPOSE_KEY  <^a/A/>,<^a/E/>,,<^xc6> ; order sensitive
        VC$COMPOSE_KEY  <^a/A/>,<^a/^/>,,<^xc2>
        VC$COMPOSE_KEY  <^a/A/>,<^a/_/>,,<^xaa>
    .
    .
    .
        VC$COMPOSE_KEYEND  COMPOSE3_TBL_LEN  ; end the table
                                             ; and determine its
                                             ; length
    .
    .
    .
```

2.5 Using a Pointer Device

The drivers are designed to detect two pointer-related conditions:

- Pointer movement
- Pointer button transition

The QIO interface permits you to associate regions of the screen with action ASTs that the driver fires whenever it detects pointer movement or a pointer button transition (clicking up or down). The action ASTs are application dependent and permit you to perform such screen manipulation as highlighting a menu when the pointer moves into it, or performing an action once a menu item is selected.

As described in Chapter 1, the driver uses separate lists to keep track of pointer movement and button transitions. The following sections describe how to create list entries for pointer movement and button transitions.

Creating a Pointer Movement Entry

To create a pointer movement entry, use the Enable Pointer Movement QIO. When you create a pointer movement entry, you may specify:

- The address of a four-longword AST specification block. The four longwords contain
  1. The address of an AST routine that determines what action to take when movement is detected within the specified region.
  2. An optional user-defined AST parameter that is delivered to the AST.
  3. Access mode at which to deliver the AST (maximized with the current access mode).
  4. The address of a longword at which the driver stores the new cursor position, so that it is accessible to the application.

     The low-order word of the longword holds the X pixel position and ranges from 0 to 1023, where 0 is the left side of the screen. The high-order word holds the Y pixel position and ranges from 0 to 863, where 0 is the bottom of the screen.
- The address of an AST routine that is invoked whenever the pointer *exits* the specified region. Typically, this AST performs any necessary cleanup actions, for example, turning off a region highlighted by the action AST.
- Address of a screen rectangle values block that describes the region to be associated with the ASTs.

By default, an entry is placed at the top of the list. However, an application may determine the position of the entry in the list by specifying optional modifiers to the QIO. The modifiers can be used to:

- Place the entry last on the list (QV_LAST)
- Delete the entry (QV_DELETE)

Whenever you move the pointer (mouse, stylus, or puck), the driver checks the pointer position against the region descriptors of the pointer movement list entries. When the driver finds an entry whose region descriptor includes the current pointer position, the driver fires the action AST associated with that entry.

If you specified an exit AST, the driver fires that AST when it detects that the pointer position is no longer within the specified region.

If two regions overlap, the first one on the list gets the AST.

The following example shows how a pointer motion AST could be programmed. (Appendix D of this manual shows this example in the context of a complete applications program.)

```
    .
    .
    .
10$:    MOVL    #IO$C_QV_MOUSEMOV,R0    ; enable pointer motion
        $QIOW_S CHAN=MOUSE_CHAN,-       ; channel
                FUNC=#IO$_SETMODE,-     ; QIO function code
                P1=(R0),-               ; driver function code
                P2=#MOUSE_BLOCK,-       ; associated AST block
                P6=#MOUSE_REGION        : associated region
        BLBS    R0,20$                  ; no error if set
        BRW     ERROR
20$:    RSB

MOUSE_BLOCK:                            ; pointer region AST
                                        ; specification block
        .LONG   MOUSE_AST               ; AST address
        .LONG   MOUSE_ACK               ; AST parameter
        .LONG   0                       ; access mode
        .LONG   MOUSE_XY                ; new pointer cursor position
                                        ; storage

MOUSE_REGION:                           ; pointer region
        .LONG   400                     ; lower left corner
        .LONG   400
        .LONG   800                     ; upper right corner
        .LONG   800
    .
    .
    .
```

Creating a Pointer Button Transition Entry

To create a pointer button transition entry, use the Enable Button Transition QIO. When you create a button transition entry, you may specify:

- The address of a four-longword AST specification block. The four longwords contain

    1. The address of an AST routine that determines what action to take when a transition is detected within the specified region.

       If no AST routine is specified, input (the button transition) will be stored in the typeahead buffer, and delivered either when an AST region is declared using the same channel or when a Get Next Input Token QIO is issued using the same channel.

2. An optional user-defined AST parameter that is delivered to the AST.
3. Access mode at which to deliver the AST (maximized with the current access mode).
4. The address of a longword at which the driver stores the button transition value (token) when the AST service routine is called.

   Your application uses this longword to determine which pointer button has undergone a transition. The button transition value (the token) is a decimal value indicating which button was activated; the transition values are 400, 401, 402, and 403. The values are assigned to the pointer buttons sequentially starting with the select button, which is always 400. The driver stores the token in the low-order word of the longword. Bit 15 of the high-order longword determines whether the transition is up or down; 1 equals down, 0 equals up.

   The rest of the high-order word contains more control information that can be used to determine if the Shift, Control, or Lock keys are depressed. You can use these keys as meta-keys (keys used in combination). Bit 14 corresponds to the Shift key, bit 13 corresponds to the Control key, and bit 12 corresponds to the Lock key. When the bit is set, the key is down.

- The address of a pointer button characteristics block which determines whether delivery of subsequent transitions depends on all the buttons being in the up position. By default, the specified button transition AST gets all transitions until all buttons are returned to the up position. (See the description of this data structure in Appendix A.)
- Address of a screen rectangle values block that describes the region to be associated with the button transition AST.

By default, an entry is placed at the top of the list. However, an application may determine the position of the entry in the list by specifying optional modifiers to the QIO. The modifiers can be used to:

- Place the entry last on the list (QV_LAST)
- Delete the entry (QV_DELETE)
- Purge the typeahead buffer (QV_PURG_TAH)

If a button changes state (is clicked up or down), the driver checks the pointer position against the region descriptors of the button list entries. When the driver finds a region descriptor which contains the current pointer position, it fires the associated button transition AST.

If two regions overlap the first one on the list gets the AST.

The following example shows the typical programming and use of pointer button ASTs. (Appendix D of this manual shows this example in the context of a complete applications program.)

```
    .
    .
    .
SET_BUTTONAST:
        $ASSIGN_S   DEVNAM=WS_DEVNAM,- ; assign channel using
                    CHAN=BUT_CHAN      ;  logical name and
                                       ;  channel number
        BLBS    R0,10$                 ; no error if set
        BRW     ERROR                  ; error

10$:    MOVL    #IO$C_QV_ENABUTTON,R0  ; enable button trans.
        $QIOW_S CHAN=BUT_CHAN,-        ; channel
                FUNC=#IO$_SETMODE,-    ; QIO function code
                P1=(R0),-              ; driver function code
                P2=#BUT_BLOCK,-        ; associated AST block
                P6=#BUT_REGION         ; associated region
        BLBS    R0,20$                 ; no error if set
        BRW     ERROR
20$:    RSB
BUT_BLOCK:                             ; button transition AST
                                       ; specification block
        .LONG   BUT_AST                ; AST address
        .LONG   0                      ; AST parameter
        .LONG   0                      ; access mode
        .LONG   BUTTON                 ; button information
                                       ; longword

BUT_REGION:                            ; associated region
        .LONG   20                     ; lower left corner
        .LONG   20
        .LONG   300                    ; upper right corner
        .LONG   300
    .
    .
    .
```

2.6 Using the Typeahead Buffer

In the case of keyboard input, pointer movement, and button transitions, the driver accepts three kinds of input: character input, pointer position, or button transition value. Often input is received faster than an application can (or chooses to) process it. When this happens the character and button information is stored in the *typeahead buffer.* (Pointer movement inputs the new pointer position, but if the input cannot be delivered, it is ignored, not buffered.)

The typeahead buffer is part of each entry on the list. It is 128 bytes long, so you are able to buffer 32 input tokens (each token is four bytes long).

2.6.1 Getting Input from the Typeahead Buffer

You can get input from the typeahead buffer in two ways:

- Associate a repeating AST with the entry when it is enabled to continuously process buffered input. (In the case of keyboards, you may also associate a repeating AST when you modify the keyboard.)
- Issue a Get Next Input Token QIO to process a single input token from the buffer (this QIO may optionally have an AST associated with it — the input is delivered in the IOSB block in either case). This type of single-token processing is referred to as a "one shot".

Once a repeating AST is associated with an entry, any attempt to issue subsequent one shots on that entry will return an error because the results are unpredictable. If you enable an entry without an associated AST you can issue one shots to process the buffered data one character at a time. You can associate a repeating AST with an entry at any time, by reenabling the entry (or for keyboards, modifying the keyboard). However, any outstanding one shots will be processed first.

Note that QIO modifiers permit you to purge the typeahead buffer. If you delete an entry, and the typeahead buffer is not empty, the deletion is deferred until the typeahead buffer is empty. If an application wants to immediately delete an entry, it must first purge the buffer.

2.7 Intercepting Input

As noted above, you can issue one shots on a channel that currently has no repeating AST associated with it. To "intercept" input, disable the associated AST (by reenabling the entry without the AST specification) and then issue one shots. (Note that for a keyboard you can disable an AST by modifying the keyboard instead of reenabling it.) At a later point you may reenable the repeating AST.

To intercept the input for an entry on a list, issue the Get Next Input Token QIO specifying the type of input token (IOC_QV_ENAKB, IOC_QV_MOUSEMOV, or IO$C_ENABUTTON), and the channel with which the entry is associated.

One shots are also useful when you wish to process input from within an AST (ASTs cannot be delivered in this case). An application can rely on the fact that a one shot with no associated AST will deliver the input to the IOSB. By using an event flag and the IOSB, the application can process the typeahead buffer one character at a time from within the AST.

The example in Appendix D contains instances of intercepting input.

2.8 Defining Cursor Patterns

The QIO interface also permits you to associate a region of the screen with a specific cursor pattern. Using this feature, you may change the shape and size of the cursor to reflect a change in functionality; for example, an editing cursor may take one shape while a menu selection cursor takes another. Again, the driver maintains an entry list to keep track of cursor patterns.

To create a cursor pattern entry, use the Define Pointer Cursor Pattern QIO. When you create a cursor pattern entry, you may specify:

- The address of a bitmap image for the new cursor pattern. This is either a 16-word array or, on multiplane cursor systems, a 32-word array. The QVB contains a field that determines whether you have a one- or multiplane system, use the Get System Information QIO to access this field. The following section describes multiplane cursor patterns.
- The address of a longword to contain a new cursor position. This optional parameter permits you to reposition the cursor.
- The address of the cursor hot spot definition. The hot spot is the one position within the bitmap image of the cursor that is the actual cursor position.
- Cursor style. This value defines how the cursor appears against the background of the screen. (It is ignored on multiplane cursor systems.)

- Address of a screen rectangle values block that describes the region to be associated with the cursor pattern.

By default, an entry is placed at the top of the list. However, an application may determine the position of the entry in the list by specifying optional modifiers to the QIO. The modifiers can be used to

- Place the entry last on the list (QV_LAST)
- Delete the entry (QV_DELETE)

As the pointer is moved, the driver checks the pointer position against the region descriptors of the cursor pattern list entries. When the driver finds a region descriptor which contains the current pointer position, it changes the cursor pattern to the one associated with the region.

The following example shows the typical assignment of a single-plane cursor region pattern. (Appendix D of this manual shows this example in the context of a complete applications program.)

```
.
.
.
QV$CURSOR1:                                    ; initial 16 x 16 cursor
                                               ; "solid" pattern
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
    .
    .
    .
```

```
REGION1:                                    ; cursor region 1
        .LONG   20                          ; lower left corner
        .LONG   20
        .LONG   300                         ; upper right corner
        .LONG   300
    .
    .
    .
20$:    MOVL    #IO$C_QV_SETCURSOR,R0       ; define cursor 1
        $QIOW_S CHAN=CUR_CHAN1,-            ; channel
                FUNC=#IO$_SETMODE,-         ; QIO function code
                P1 =(R0),-                  ; driver function code
                P2=#QV$CURSOR1,-            ; cursor pattern
                P6=#REGION1                 ; associated region
        BLBS    R0,30$                      ; no error if set
        BRW     ERROR
    .
    .
    .
```

2.8.1 Multiplane Cursor Patterns

If your system uses a multiplane cursor, you may specify a 32-word array as a cursor pattern. Currently, multiplane cursors consist of two planes. Typically, you use two planes to prevent the cursor from disappearing when it is moved over varying backgrounds. To understand how the two planes work think of the 32-bit array as two 16-word arrays, array A and array B.

The bit pattern in array A is determined in the following way.

- A 1 in a bit indicates that the corresponding pixel be filled.
- A 0 in a bit indicates that whatever is on the screen at the corresponding pixel should show through (remember, the cursor is overlaid on the screen).

The bit pattern in array B uses the bits set to 0 in array A as a mask: those corresponding bits are ignored in array B. The remaining bit pattern in array B is determined in the following way:

- A 1 in a bit indicates that the corresponding pixel be filled with the background color.
- A 0 in a bit indicates that the corresponding pixel be filled with the foreground color.

2.9 Using an Alternate Windowing System

To provide greater flexibility the QVSS driver supports a single private graphics application in addition to the default, VWS-supplied, windowing package. That is, you can write an alternate windowing application that takes complete control of video memory and does not depend on VWS-supplied window or graphics services.

To enable alternate windowing take the following step:

1. Modify the command procedure SYSTARTUP.COM to define the logical name UIS$WS_ALTAPPL to "TRUE". This definition must occur before the STARTVWS.COM command procedure is invoked.

This instructs the driver to reserve one-half of video memory for a private application. When requested by the private application this part of video memory is mapped to the screen and made available to the application. All set mode functions issued by the application relate only to its *private* video memory. A user interface key (F3) on the keyboard allows an operator to switch between windowing systems dynamically.

Note that private applications are device dependent and that only one private application may be active at a time.

2.10 Drawing to the QVSS Screen

To draw to the screen using the QVSS driver, take the following steps:

1. Access the QVB.
2. Manipulate bits in video memory.
3. Map the manipulated video memory to the screen.

Section 2.2 describes how to access the QVB. The following sections describe how to manipulate bits and map video memory to the screen.

2.10.1 Manipulating Bits in Video Memory

A QVSS application "draws" by directly setting bits in video memory. To access video memory use the QVB$L_MAIN_VIDEOADDR address in the QVB. It is the responsibility of the application to determine how to offset into video memory. When manipulating memory, keep the following in mind:

- Each scanline of video memory is 1024 bits (128 bytes) wide.
- There are 1024 scanlines in memory. (Note that if an alternate windowing system is used, the accessible number of scanlines is effectively halved.)

2.10.2 Mapping Video Memory to the Screen

To map a scanline in memory to the screen, you must load an entry in the scanline map. The scanline map consists of word-length entries whose positions in the map correspond to line positions on the screen and whose contents are indexes of scanline positions in video memory. The index of scanlines in memory starts at zero and is simply incremented by one for each scanline.

Mapping with an Alternate Windowing System

This scheme is straightforward unless you are using an alternate windowing system, in which case memory is split in half and shared by two systems. You must ensure that you are mapping the correct portion of memory by calculating the correct scanline base in video memory. To obtain the correct scanline base, do the following:

1. Subtract the QVB_$L_VIDEOADDR address from the QVB$L_MAIN_ VIDEOADDR address.
2. Divide the result by 128 (number of bytes in a scanline).

Add the base to any scanline index before inserting it as an entry in the scanline map.

2.11 Creating a QDSS Viewport

The QDSS driver performs all operations to the screen using viewports. If your application is not using the UIS windowing interface, it must either create a viewport before attempting to write to the screen, or use the systemwide viewport (the entire screen). Example 2–5 demonstrates how to access the systemwide viewport.

To create a viewport an application must perform the following steps:

1. Assign the viewport a channel.
2. Get a viewport ID.
3. Define the location and size of the viewport.
4. Start the viewport.

The following sections describe how to perform each of these steps.

2.11.1 Assigning a Viewport Channel

To obtain a unique channel for a viewport, use the $ASSIGN system routine. The actual association of the viewport with the channel occurs when the application gets a viewport ID for the viewport.

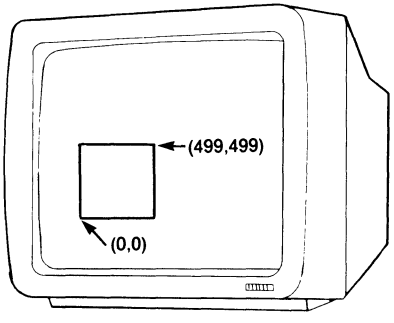
2.11.2 Getting a Viewport ID

To get a viewport ID, use the Get Viewport ID QIO. One parameter of this QIO specifies the address of the longword to receive the ID. The ID stored at that address is used to identify which viewport is the object of all subsequent operations. The channel the application specifies in this QIO is associated with the viewport.

2.11.3 Defining a Viewport

As described in Chapter 1, a viewport is defined by one or a number of rectangular update regions. Update regions are defined in buffers known as Update Region Definition (URD) buffers. Each URD buffer contains coordinate information that defines the dimensions, in pixels, of a rectangle, and its location relative to the base of QDSS memory either onscreen or offscreen. (See Appendix B for detailed information about this data structure.) A viewport may be defined by one or more URDs. Figure 2–7 illustrates a 500- by 500-pixel viewport defined by a single URD and displays the contents of the associated definition buffer.

Figure 2–7 Viewport and Update Region Definition Buffer



X	Y	
0	0	Lower Left Corner (URC)
499	499	Upper Right Corner (VRC)
50	100	Absolute Base (ADC)

ZK-5348-86

Note that the base is given in absolute coordinates while the two defining corners of the viewport are given in viewport relative coordinates. This becomes important when a viewport is divided into a number of rectangles and some (occluded) rectangles are stored in offscreen memory. Drawing operations use the relative coordinates to perform drawing even when rectangles are not visible on the screen.

To define a viewport:

- Allocate and initialize one or more URD buffers that describe the viewport's size and relative position.
- Call the Define Viewport Region QIO once for each *viewport*, passing the URD (or array of URDs if the viewport is more than one rectangle).

Parameters of the Define Viewport Region QIO specify the address of the viewport definition buffer, the length of that buffer, and the viewport ID.

You can redefine a viewport at any time by reinvoking the Define Viewport Region QIO with new coordinate information and the same channel and viewport ID.

2.11.4 Starting the Viewport

When you define a viewport, it is in a "stopped" state. To permit operations to the viewport, you must explicitly start the viewport request queue. To start the viewport request queue, use the Start Request Queue QIO.

The following example creates and starts a viewport that is 100 pixels square, and has its lower left corner at the absolute device coordinate (10,10). Note that in FORTRAN you must include the IODEF library in order to access the QIO function codes.

Example 2–2 Creating a Viewport

```
PROGRAM CREATE_VIEWPORT
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

INTEGER*2 CHAN_VP1,
2         CHAN_VP2

! Declare URDs
INTEGER*2 URD1_VP1(6),
2         URD1_VP2(6)

! Load URD1_VP1 buffer
URD1_VP1(1) = 0          ! lower left corner
URD1_VP1(2) = 0
URD1_VP1(3) = 99         ! upper right corner
URD1_VP1(4) = 99
URD1_VP1(5) = 10         ! absolute coordinate base
URD1_VP1(6) = 10

! define and start VP1
CALL VIEWPORT (URD1_VP1, CHAN_VP1, VP1_ID)



! ******************
! Viewport Subroutine
! ******************

SUBROUTINE VIEWPORT (VP_URD, VP_CHANNEL, VIEWPORT_ID)

IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

INTEGER*2 VP_CHANNEL

! Obtain a channel for the viewport
CALL SYS$ASSIGN ('SYS$WORKSTATION',     ! device name
2                VP_CHANNEL,,)          ! channel
```

(Continued on next page)

Example 2-2 (Cont.) Creating a Viewport

```
! Get a viewport ID
CODE = IO$_SENSEMODE
STATUS = SYS$QIOW (,
2                   %VAL(VP_CHANNEL),              ! channel
2                   %VAL(CODE),                    ! QIO function code
2                   ,,,
2                   %VAL(IO$C_QD_GET_VIEWPORT_ID),
2                   VIEWPORT_ID,                   ! address of ID buffer
2                   %VAL(4),                       ! VP ID buffer length
2                   ,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Define the Viewport Region
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2                   %VAL(VP_CHANNEL),              ! channel
2                   %VAL(CODE),                    ! QIO function code
2                   ,,,
2                   %VAL(IO$C_QD_SET_VIEWPORT_REGIONS),
2                   VP_URD,                        ! address of URD buffer
2                   %VAL(URD$C_LENGTH),            ! length of URD buffer
2                   %VAL(VIEWPORT_ID),,)           ! address of VP ID
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Start the Viewport
STATUS = SYS$QIOW (,
2                   %VAL(VP_CHANNEL),     ! channel
2                   %VAL(CODE),           ! QIO function code
2                    ,,,
2                   %VAL(IO$C_QD_START),  ! QD function code
2                   %VAL(VIEWPORT_ID),    ! address of ID buffer
2                   ,,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

RETURN
END
```

2.12 Drawing with the QDSS Driver

Unlike the QVSS driver, the QDSS driver has drawing capabilities. The QDSS driver uses data structures known as drawing operation primitives (DOPs) to perform drawing operations. Your application loads a DOP with all the information necessary for the hardware to perform a drawing operation. Typically, a DOP contains the type of operation to perform (that is, draw a line, draw text, and so on), the number of times to perform it, and any coordinates needed to perform the operation.

As described in Chapter 1, you must allocate storage for DOPs, insert DOPs on the request queue to execute them, and reuse the storage using the return queue. If your application is using the UIS windowing environment, you can perform all three of these functions using the UISDC routines. However, if your application is *not* using the UIS windowing environment, the *application* must manage DOP storage and insert DOPs on the request queue.

Chapter 5 describes how to perform drawing using DOPs. It describes:

- The structure of DOPs
- How to use the UISDC interface
- How to allocate and execute DOPs without the UISDC interface

2.13 Using Bitmaps

Although the QDSS driver does not support direct manipulation of the onscreen bitmap, it does permit you to copy bitmap images from processor memory to onscreen and offscreen memory and from on and offscreen memory to processor memory.

The driver provides the following QIOs for manipulating bitmaps:

Write Bitmap — Copies a bitmap from processor memory to QDSS screen memory and performs bitmap-to-bitmap transfers (onscreen-to-offscreen and offscreen-to-onscreen).

Read Bitmap — Copies a bitmap from QDSS memory to processor memory and performs bitmap-to-bitmap transfers (onscreen-to-offscreen and offscreen-to-onscreen).

Load Bitmap — Loads a bitmap to be used by a text or fill pattern drawing operation from processor memory into the reserved bitmap area of offscreen memory (see Figure 1–4). This QIO returns a bitmap ID that the DOPs use to reference the bitmap. Bitmaps loaded by this QIO must follow certain criteria; see the QDSS QIO section for details. The UISDC interface also provides a Load Bitmap function. (See Chapter 5.)

To draw an image, build it in processor memory, then load it into QDSS memory using the Write Bitmap QIO.

To store an image in processor memory, copy the bitmap from QDSS memory to processor memory, using the Read Bitmap QIO. (This is what is done for occluded viewport regions when offscreen memory becomes full; see Section 2.18.)

To load a bitmap for use with a DOP, use the Load Bitmap QIO.

Example 2-3 illustrates the use of the Write Bitmap QIO to copy a region from onscreen memory into offscreen memory (a bitmap-to-bitmap transfer).

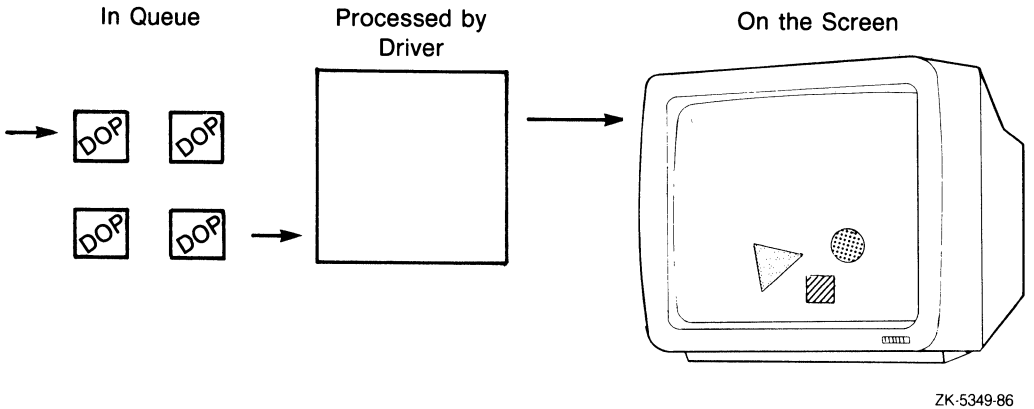
2.14 Synchronizing Viewport Activity

Because of the asynchronous way DOPs are queued for processing and execution, you must take special actions if you wish to synchronize activity on a viewport.

Figure 2-8 illustrates the three states that DOPs may have. They may be:

- In the queue, waiting to be processed
- Currently being processed by the driver
- Completed and on the screen

Figure 2-8 Synchronizing Viewport Activity



At any given time, the driver may be processing a number of DOPs. To synchronize activity, you must manipulate the queue *and* be aware of whether the driver is currently processing DOPs.

QIOs

The QIO interface provides the following QIOs to use for synchronization:

Stop Request Queue — Immediately halts the processing of the request queue and waits for whatever is currently being processed to complete before returning.

Start Request Queue — Restarts processing on a stopped request queue.

Suspend Request Queue — Immediately halts the processing of the request queue *but does not wait* for whatever is currently being processed to complete before returning.

Resume Request Queue — Resumes processing on a suspended request queue.

Hold Viewport Activity — Does not permit any viewport except the systemwide viewport to write to the screen (processing actually continues.)

Release Hold — Releases the hold on viewport activity.

Insert DOP — Permits an application to insert a DOP on the request queue and waits for completion — essentially performing a synchronous DOP.

DOPs

In addition to these QIOs, the request queue interface permits you to submit the following DOPs:

Stop Viewport Activity — Halts the queue and waits for any DOPs currently processing to complete. This differs from the Stop Request Queue QIO in that all DOPs inserted before this one are guaranteed to execute before the queue is stopped.

Start Viewport Activity — Restarts processing on a stopped request queue.

Suspend Viewport Activity — Halts the queue *but does not wait* for any DOPs currently processing to complete. This differs from the Suspend Viewport Activity QIO in that all DOPs inserted before this one are guaranteed to execute before the queue is stopped.

Resume Viewport Activity — Resumes processing on a suspended request queue.

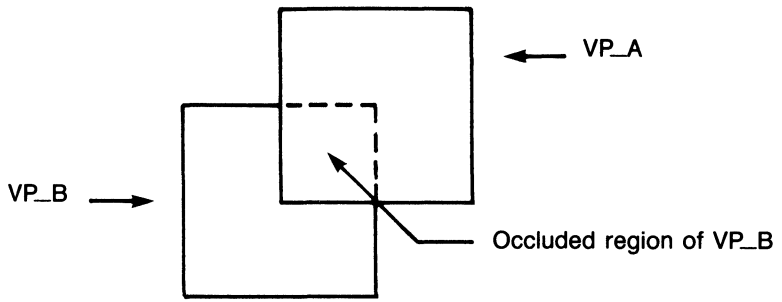
A Stop also differs from a Suspend in that a Stop issued on a stopped request queue waits for the queue to restart, then takes effect — while a Suspend issued on a suspended viewport is ignored. This makes the Stop useful for synchronizing multiprocess windowing activity on a single viewport. A process can guarantee that no other process is accessing the viewport by issuing a Stop before attempting any windowing activity (redefining URDs, and so forth). When control returns from the Stop, the process knows that no other process' DOPs can execute on the viewport and any DOPs that were processing have completed.

Note that if you issue a Stop Request Queue QIO to an already stopped viewport, the QIO will not complete until the viewport is started by an AST routine or another process. *However,* if you issue a Stop Request Queue QIO to a *suspended* viewport, the Stop Request Queue QIO *will* complete.

2.15 Handling Occlusion

In multiviewport systems, there may be instances in which two or more viewports overlap. The area of a viewport overlapped by another viewport is said to be occluded. Figure 2–9 illustrates one viewport (VP_A) occluding another (VP_B).

Figure 2–9 Occluded Viewport



Only one viewport can display the overlapped area of the bitmap at a given time. If your application is designed to permit occlusion, it must be able to handle any operations directed to a viewport region that is occluded. You do this by moving the occluded region of the viewport into offscreen memory and performing any operations directed to it in offscreen memory. If, at a later time, that portion of the screen becomes available for display, you can then copy the up-to-date region back to the screen. (This is referred to as "popping" the viewport.) The following sections describe how to handle a simple case of occlusion.

2.15.1 Redefining Viewports

An application uses the update region definition buffers to handle occlusion. A viewport that is originally defined as one rectangle, using a single URD, can be redefined as a number of rectangles (viewport regions) using a URD for each rectangle. The URDs provide both the absolute position of the rectangle in QDSS memory and the viewport-relative coordinates of the rectangles in relation to each other.

Use the Define Viewport Region QIO to redefine a viewport. You can redefine an occluded region to be in offscreen memory (using a negative Y coordinate); because the drawing operations use viewport-relative coordinates, the drawing is performed properly. You must ensure that the negative Y coordinate you use falls within the range of the free_1 area of offscreen memory (see Figure 1–4).

Figure 2–10 illustrates the partitioning of an occluded viewport. In this example, the viewport is divided into three rectangles (A, B, and C). The minimum, maximum, and base (X,Y) coordinate pairs are stored in three definition buffers.

The base coordinates of each accessible rectangle are in absolute device coordinates which are relative to the base of display memory (0,0). A base value with two positive coordinates indicates that the rectangle is in onscreen display memory. A base value with a negative Y coordinate and a positive X coordinate indicates that the rectangle is in offscreen memory. A base value of (-1,-1), for instance, indicates that the rectangle is on the deferred queue; see Section 2.18 for information about the deferred queue. Note that rectangle A is redefined to be in offscreen memory.

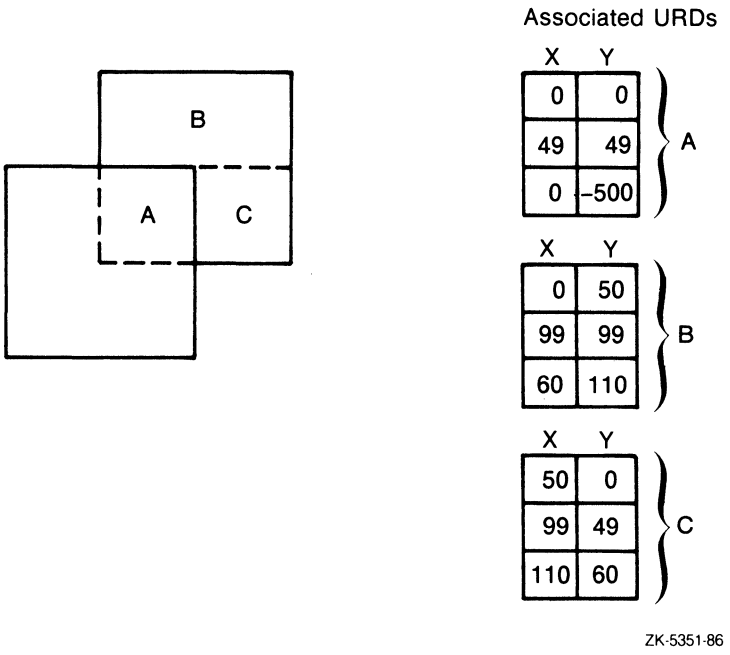
2.15.2 Securing Exclusive Access to the Bitmap

An application secures exclusive access to the bitmap to guarantee that two or more overlapping viewports do not attempt to write to the same piece of the display simultaneously. Before creating a viewport, your application should determine whether another viewport already exists in the area of the screen in which the viewport will be created. (Keeping track of where each viewport is on the screen is a bookkeeping function that is the responsibility of the application.)

To secure exclusive access to a viewport bitmap:

1. Stop activity on the existing viewport using the Stop Request Queue QIO. This must be done to ensure a known state for the subsequent steps.
2. Redefine the regions of the existing viewport using the Define Viewport Region QIO.
3. Copy the to-be-occluded region of the existing viewport to offscreen memory, using the Write Bitmap or Read Bitmap QIO.

Figure 2-10 Redefining Viewports with URDs



4. Update the URD definition of the existing viewport to reflect its new state. Redefine the occluded region to be offscreen by specifying a negative Y coordinate in the base value of the URD. (The negative Y value must fall within the range of the free area of offscreen memory shown in Figure 1–4). Remember, drawing can still be performed to offscreen memory.
5. Restart the viewport using the Start Request Queue QIO.

Once this is done, you can create the new viewport on screen and start drawing operations on it.

The following example occludes one viewport with another. Exclusive access to the bitmap is guaranteed before the second viewport is created. The occluded region of the existing viewport is copied into the offscreen memory free area at (0,-200). Note that the transfer parameter block (TPB) is loaded using the predefined structure in the VWSSYSDEF file.

Example 2–3 Securing Bitmap Access

```
PROGRAM CREATE_VIEWPORT
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

INTEGER*2 CHAN_VP1,
2         CHAN_VP2

! Declare TPB
INTEGER*2 TPB(13)

! Declare URDs
INTEGER*2 URD1_VP1(6),
2         URD1_VP2(6)

! Load URD1_VP1 buffer
URD1_VP1(1) = 0         ! lower left corner
URD1_VP1(2) = 0
URD1_VP1(3) = 99        ! upper right corner
URD1_VP1(4) = 99
URD1_VP1(5) = 10        ! absolute coordinate base
URD1_VP1(6) = 10

! Load URD1_VP2 buffer
URD1_VP2(1) = 0         ! lower left corner
URD1_VP2(2) = 0
URD1_VP2(3) = 99        ! upper right corner
URD1_VP2(4) = 99
URD1_VP2(5) = 60        ! absolute coordinate base
URD1_VP2(6) = 60

! Define and start VP1
CALL VIEWPORT (URD1_VP1, CHAN_VP1, VP1_ID)
.
.
.

! Stop VP1
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2             %VAL(CHAN_VP1),           ! channel
2             %VAL(CODE),               ! QIO function code
2             ,,,
2             %VAL(IO$C_QD_STOP),       ! QD function code
2             %VAL(VP1_ID),             ! address of ID buffer
2             ,,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

(Continued on next page)

Example 2-3 (Cont.) Securing Bitmap Access

```
! Load TPB for occluded rectangle
CALL LOAD_TPB (TPB)

! Copy occluded rectangle into offscreen memory
CODE = IO$_QDWRITE
STATUS = SYS$QIOW (,
2                   %VAL(CHAN_VP1),                ! channel
2                   %VAL(CODE),                    ! QIO function code
2                   ,,,,,,
2                   TPB,                           ! transfer block
2                   %VAL(TPB$C_BITMAP_XFR_LENGTH),)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Update regions of VP1
CALL UPDATE_REGIONS (CHAN_VP1, VP1_ID)

! Restart VP1
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2                   %VAL(CHAN_VP1),       ! channel
2                   %VAL(CODE),           ! QIO function code
2                   ,,,
2                   %VAL(IO$C_QD_START),  ! QD function code
2                   %VAL(VP1_ID),         ! address of ID buffer
2                   ,,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Define and start VP2
CALL VIEWPORT (URD1_VP2, CHAN_VP2, VP2_ID)
.
.
.
! **********************
! * LOAD TPB SUBROUTINE *
! **********************

SUBROUTINE LOAD_TPB (TPB)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ TPB
RECORD /TPB_STRUCTURE/ TPB
```

(Continued on next page)

Example 2–3 (Cont.) Securing Bitmap Access

```
! Load values
TPB.TPB$B_TYPE = TPB$C_BITMAP_XFR       ! type
TPB.TPB$B_SIZE = TPB$C_LENGTH
TPB.TPB$W_X_SOURCE = 60                 ! x of lower left corner
TPB.TPB$W_Y_SOURCE = 60                 ! y of lower left corner
TPB.TPB$W_WIDTH = 50                    ! width of source
TPB.TPB$W_HEIGHT = 50                   ! height of source
TPB.TPB$W_X_TARGET = 0                  ! x of lower left corner
TPB.TPB$W_Y_TARGET = -200               ! y of lower left corner

RETURN
END

! ****************************
! * UPDATE REGION SUBROUTINE *
! ****************************

SUBROUTINE UPDATE_REGIONS (VP_CHANNEL, VIEWPORT_ID)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

! Declare URD
INTEGER*2 URD(18)

! Load URD buffer
! First rectangle
URD(1) = 0      ! lower left corner
URD(2) = 0
URD(3) = 99     ! upper right corner
URD(4) = 49
URD(5) = 10     ! absolute coordinate base
URD(6) = 10

! Second rectangle
URD(7) = 0      ! lower left corner
URD(8) = 49
URD(9) = 49     ! upper right corner
URD(10) = 99
URD(11) = 10    ! absolute coordinate base
URD(12) = 59
! Third rectangle
URD(13) = 49    ! lower left corner
URD(14) = 49
URD(15) = 99    ! upper right corner
URD(16) = 99
URD(17) = 0     ! absolute coordinate base
URD(18) = -200  ! (offscreen)
```

(Continued on next page)

Example 2-3 (Cont.) Securing Bitmap Access

```
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2                   %VAL(VP_CHANNEL),        ! channel
2                   %VAL(CODE),             ! QIO function code
2                   ,,,
2                   %VAL(IO$C_QD_SET_VIEWPORT_REGIONS),
2                   URD,                     ! address of URD buffer
2                   %VAL(3 * URD$C_LENGTH), ! length of URD buffer
2                   %VAL(VIEWPORT_ID),,)     ! address of VP ID
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

RETURN
END
```

2.15.3 Popping an Occluded Viewport

Bringing an occluded viewport into full view onscreen is referred to as *popping* a viewport. Popping a viewport involves copying the *occluding* region into offscreen memory and the *occluded* region from offscreen memory onto the screen. To pop a viewport, an application must take the following steps:

1. Stop activity on the occluding viewport.
2. Copy the occluding region into offscreen memory.
3. Redefine the occluding viewport's URDs.
4. Restart the occluding viewport.
5. Stop activity on the occluded viewport.
6. Copy the occluded region from offscreen memory onto the screen.
7. Redefine the occluded viewport's URDs.
8. Restart the (formerly) occluded viewport.

The following example pops a viewport:

Example 2–4 Popping a Viewport

```
PROGRAM POP
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

INTEGER*2 CHAN_VP1,
2         CHAN_VP2

! Declare TPBs
INTEGER*2 TPB1(13),
2         TPB2(13),
2         TPB3(13)

! Declare URDs
INTEGER*2 URD1_VP1(6),
2         URD1_VP2(6)

! Load URD1_VP1 buffer
URD1_VP1(1) = 0          ! lower left corner
URD1_VP1(2) = 0
URD1_VP1(3) = 99         ! upper right corner
URD1_VP1(4) = 99
URD1_VP1(5) = 10         ! absolute coordinate base
URD1_VP1(6) = 10

! Load URD1_VP2 buffer
URD1_VP2(1) = 0          ! lower left corner
URD1_VP2(2) = 0
URD1_VP2(3) = 99         ! upper right corner
URD1_VP2(4) = 99
URD1_VP2(5) = 60         ! absolute coordinate base
URD1_VP2(6) = 60

! Define and start two overlapping viewports
.
.
.
```

(Continued on next page)

Example 2-4 (Cont.) Popping a Viewport

```
! Stop VP2 (the occluding viewport)
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2             %VAL(CHAN_VP2),          ! channel
2             %VAL(CODE),              ! QIO function code
2             ,,,
2             %VAL(IO$C_QD_STOP),      ! QD function code
2             %VAL(VP2_ID),            ! address of ID buffer
2             ,,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Load TPB for occluded rectangle
CALL LOAD_TPB2 (TPB2)

! Copy occluding region into offscreen memory
CODE = IO$_QDWRITE
STATUS = SYS$QIOW (,
2                 %VAL(CHAN_VP2),      ! channel
2                 %VAL(CODE),          ! QIO function code
2                 ,,,,,,
2                 TPB2,                ! transfer block
2                 %VAL(TPB$C_BITMAP_XFR_LENGTH),)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Update regions of VP2
KEY = 2
CALL UPDATE_REGIONS (CHAN_VP2, VP2_ID, KEY)

! Restart VP2
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2                 %VAL(CHAN_VP2),      ! channel
2                 %VAL(CODE),          ! QIO function code
2                 ,,,
2                 %VAL(IO$C_QD_START), ! QD function code
2                 %VAL(VP2_ID),        ! address of ID buffer
2                 ,,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

(Continued on next page)

Example 2-4 (Cont.) Popping a Viewport

```
! Stop VP1 (the occluded viewport)
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2             %VAL(CHAN_VP1),           ! channel
2             %VAL(CODE),               ! QIO function code
2             ,,,
2             %VAL(IO$C_QD_STOP),       ! QD function code
2             %VAL(VP1_ID),             ! address of ID buffer
2             ,,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Load TPB for occluded rectangle
CALL LOAD_TPB3 (TPB3)

! Copy offscreen rectangle into screen memory
CODE = IO$_QDWRITE
STATUS = SYS$QIOW (,
2                 %VAL(CHAN_VP1),       ! channel
2                 %VAL(CODE),           ! QIO function code
2                 ,,,,,,
2                 TPB3,                 ! transfer block
2                 %VAL(TPB$C_BITMAP_XFR_LENGTH),)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Update regions of VP1
KEY = 3
CALL UPDATE_REGIONS (CHAN_VP1, VP1_ID, KEY)

! Restart VP1
CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2                 %VAL(CHAN_VP1),       ! channel
2                 %VAL(CODE),           ! QIO function code
2                 ,,,
2                 %VAL(IO$C_QD_START),  ! QD function code
2                 %VAL(VP1_ID),         ! address of ID buffer
2                 ,,,)
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

(Continued on next page)

Example 2-4 (Cont.) Popping a Viewport

```
.
.
.
! ************************
! * LOAD TPB2 SUBROUTINE *  2
! ************************

SUBROUTINE LOAD_TPB2 (TPB)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ TPB
RECORD /TPB_STRUCTURE/ TPB

! Load values
TPB.TPB$B_TYPE = TPB$C_BITMAP_XFR       ! type
TPB.TPB$B_SIZE = TPB$C_LENGTH
TPB.TPB$W_X_SOURCE = 60                 ! x of lower left corner
TPB.TPB$W_Y_SOURCE = 60                 ! y of lower left corner
TPB.TPB$W_WIDTH = 50                    ! width of source
TPB.TPB$W_HEIGHT = 50                   ! height of source
TPB.TPB$W_X_TARGET = 0                  ! x of lower left corner
TPB.TPB$W_Y_TARGET = -500               ! y of lower left corner

RETURN
END

! ************************
! * LOAD TPB3 SUBROUTINE *  3
! ************************

SUBROUTINE LOAD_TPB3 (TPB)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ TPB
RECORD /TPB_STRUCTURE/ TPB
```

(Continued on next page)

Example 2–4 (Cont.) Popping a Viewport

```
! Load values
TPB.TPB$B_TYPE = TPB$C_BITMAP_XFR       ! type
TPB.TPB$B_SIZE = TPB$C_LENGTH
TPB.TPB$W_X_SOURCE = 0                  ! x of lower left corner
TPB.TPB$W_Y_SOURCE = -200               ! y of lower left corner
TPB.TPB$W_WIDTH = 50                    ! width of source
TPB.TPB$W_HEIGHT = 50                   ! height of source
TPB.TPB$W_X_TARGET = 60                 ! x of lower left corner
TPB.TPB$W_Y_TARGET = 60                 ! y of lower left corner

RETURN
END

! ***************************
! * UPDATE REGION SUBROUTINE *
! ***************************

SUBROUTINE UPDATE_REGIONS (VP_CHANNEL, VIEWPORT_ID, KEY)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

! Declare URD
INTEGER*2 URD(18)

! Assume long URD
URD_LENGTH = (3 * URD$C_LENGTH)

! Key determines which URD is loaded
IF (KEY .EQ. 1) THEN
.
.
.
ELSE IF (KEY .EQ. 2) THEN

   ! Redefine VP2 for occlusion
   ! First rectangle
   URD(1) = 0           ! lower left corner
   URD(2) = 0
   URD(3) = 49          ! upper right corner
   URD(4) = 49
   URD(5) = 0           ! absolute coordinate base
   URD(6) = -500        ! (offscreen)
```

(Continued on next page)

Example 2-4 (Cont.) Popping a Viewport

```
        ! Second rectangle
        URD(7) = 0              ! lower left corner
        URD(8) = 50
        URD(9) = 99             ! upper right corner
        URD(10) = 99
        URD(11) = 60            ! absolute coordinate base
        URD(12) = 110
        ! Third rectangle
        URD(13) = 50            ! lower left corner
        URD(14) = 0
        URD(15) = 99            ! upper right corner
        URD(16) = 49
        URD(17) = 110           ! absolute coordinate base
        URD(18) = 60

    ELSE IF (KEY .EQ. 3) THEN

        ! Redefine VP1 for pop
        URD(1) = 0    ! lower left corner
        URD(2) = 0
        URD(3) = 99   ! upper right corner
        URD(4) = 99
        URD(5) = 10   ! absolute coordinate base
        URD(6) = 10
        URD_LENGTH = URD$C_LENGTH

    ELSE

.
.
.

    END IF

    CODE = IO$_SETMODE
    STATUS = SYS$QIOW (,
    2                   %VAL(VP_CHANNEL),       ! channel
    2                   %VAL(CODE),             ! QIO function code
    2                   ,,,
    2                   %VAL(IO$C_QD_SET_VIEWPORT_REGIONS),
    2                   URD,                    ! address of URD buffer
    2                   %VAL(URD_LENGTH),       ! length of URD buffer
    2                   %VAL(VIEWPORT_ID),,)    ! address of VP ID
    IF (STATUS .NE. 1) THEN
       CALL LIB$SIGNAL (%VAL (STATUS))
    END IF

    RETURN
    END
```

2.16 Deleting a Viewport

When deleting a viewport, synchronization of activity is important. Your application must guarantee that all drawing activity to a viewport is completed *before* the viewport is deleted. Once all drawing is completed, you can deassign the associated channel to ensure that nothing else is written to the viewport. Finally, you may erase the viewport. The following sections describe each of these procedures.

2.16.1 Synchronizing Viewport Deletion

To guarantee that all drawing to a viewport is complete before deassigning a channel, an application must take the following steps:

- Issue a Stop Viewport Activity DOP *using the Insert DOP QIO* (the QDWRITE function code with the IO$M_QD_INSERT_DOP modifier). This ensures that both pending operations on the DOP request queue and those in progress are completed before the delete. The QIO is used because it will wait for the stop to occur before returning control. This is done to account for the lag time in processing DOPs. If you did not wait for completion you might delete the viewport while DOPs were on the queue.
- Deassign the associated channel, using the $DASSGN system service.

2.16.2 Erasing a Viewport

To erase a viewport, draw a rectangle of background color over the viewport, using the Fill Polygon DOP. You must use the systemwide viewport to perform this operation. (The channel of the viewport to be erased is already disassociated.) Take the following steps:

- Assign a channel for the systemwide viewport.
- Obtain the system information block using the Get System Information QIO.
- Extract the systemwide viewport ID from the system information block.
- Use the Fill Polygon DOP to draw a rectangle of background color over the viewport (see Chapter 5 for details about DOPs). The systemwide viewport ID is needed to perform this DOP on the systemwide viewport.

The following example deletes a viewport.

Example 2-5 Deleting a Viewport

```
PROGRAM DELETE_VIEWPORT
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

! Declare external macro routine
EXTERNAL DOP$INSQUE

INTEGER*2 CHAN_VP1,
2         CHAN_SYS

! Declare TPB
INTEGER*2 TPB(13)

! Declare URD
INTEGER*2 URD1_VP1(6)

! Declare QDB descriptor and buffer
INTEGER*4 QDB_DESC(2)

! Load URD1_VP1 buffer
URD1_VP1(1) = 0          ! lower left corner
URD1_VP1(2) = 0
URD1_VP1(3) = 99         ! upper right corner
URD1_VP1(4) = 99
URD1_VP1(5) = 10         ! absolute coordinate base
URD1_VP1(6) = 10

! Define and start VP1
CALL VIEWPORT (URD1_VP1, CHAN_VP1, VP1_ID)

! Draw to the viewport
.
.
.

!*********************
! Delete the Viewport
!*********************

! Synchronize the deletion
! get a Stop DOP for VP1
SIZE = (DOP$C_LENGTH)           ! calculate size
CALL GET_DOP (VP1_ID, SIZE, DOP2)

! Call the Stop subroutine
CALL STOP_VP (%VAL(DOP2),       ! DOP address, by value
2             SIZE,             ! DOP size
2             VP1_ID)           ! viewport ID
```

(Continued on next page)

Example 2–5 (Cont.) Deleting a Viewport

```
! Insert the DOP using Insert DOP QIO
CODE = (IO$_QDWRITE .OR. IO$M_QD_INSERT_DOP)
STATUS = SYS$QIOW (,
2                  %VAL(CHAN_VP1),              ! channel
2                  %VAL(CODE),,,,               ! QIO function code
2                  DOP2,                        ! DOP address
2                  %VAL(SIZE),                  ! DOP size
2                  %VAL(VP1_ID),,,)             ! VP ID
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Deassign the viewport channel
CALL SYS$DASSGN (CHAN_VP1)      ! channel

! Obtain a channel for the systemwide VP
CALL SYS$ASSIGN ('SYS$WORKSTATION',     ! device name
2                CHAN_SYS,,)            ! channel

! Get the systemwide viewport ID
! Get the QDB
CODE = IO$_SENSEMODE
STATUS = SYS$QIOW (,
2                  %VAL(CHAN_SYS),          ! channel
2                  %VAL(CODE),              ! QIO function code
2                  ,,,
2                  %VAL(IO$C_QV_GETSYS),
2                  QDB_DESC,,,,)            ! address of descriptor
IF (STATUS .NE. 1) THEN
   CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Extract the ID from the QDB
SYS_ID = EXTRACT_SYS_ID (%VAL(QDB_DESC(2))) ! pass address by value

! Get a Fill Polygon DOP
SIZE = (DOP_POLY$C_LENGTH)              ! calculate size
CALL GET_DOP (SYS_ID, SIZE, DOP3)

! Call the Fill Polygon subroutine to erase VP1 border
CALL F_POLY (%VAL(DOP3),                ! DOP address, by value
2            %VAL(DOP3+DOP$C_LENGTH),   ! var. block address
2            SIZE)                      ! DOP size

! Queue the DOP by calling a MACRO subroutine
CALL DOP$INSQUE (%VAL(DOP3),    ! DOP address, by value
2                SYS_ID)        ! viewport ID
```

(Continued on next page)

Example 2-5 (Cont.) Deleting a Viewport

```
.
.
.
! ********************
!  Get DOP Subroutine
! ********************
SUBROUTINE GET_DOP (VIEWPORT_ID, SIZE, DOP)
IMPLICIT INTEGER*4(A-Z)

! Declare external macro routine
EXTERNAL DOP$REMQUE

DOP = DOP$REMQUE (VIEWPORT_ID,
2                 SIZE)

! If none on return queue, calculate size and allocate one.
IF (DOP .EQ. 0) THEN
   CALL TEST_SIZE (%VAL(VIEWPORT_ID),    ! viewport ID > return Q
2                  SIZE)
   ! Allocate appropriate size DOP
   CALL LIB$GET_VM (SIZE,
2                   DOP)
END IF

RETURN
END

! ************************
! * TEST_SIZE SUBROUTINE *
! ************************

SUBROUTINE TEST_SIZE (REQ,SIZE)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ REQ
RECORD /REQ_STRUCTURE/ REQ

IF (SIZE .GT. REQ.REQ$W_SMALL_DOP_SIZE) THEN
   SIZE = REQ.REQ$W_LARGE_DOP_SIZE
ELSE
   SIZE = REQ.REQ$W_SMALL_DOP_SIZE
END IF

RETURN
END
```

(Continued on next page)

Example 2-5 (Cont.) Deleting a Viewport

```
.
.
.
! ****************************
! * EXTRACT_SYS_ID SUBROUTINE *
! ****************************

FUNCTION EXTRACT_SYS_ID (QDB)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ QDB
RECORD /QVB_QDSS_STRUCTURE/ QDB

EXTRACT_SYS_ID = QDB.QDB$L_SYSVP

RETURN
END

! **********************
! * STOP_VP SUBROUTINE *
! **********************

SUBROUTINE STOP_VP (DOP, SIZE, VIEWPORT_ID)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Load the Common block
DOP.DOP$W_SIZE = SIZE
DOP.DOP$W_FLAGS = 0
DOP.DOP$W_MODE = WRIT$M_NO_SRC_COMP + 10
DOP.DOP$L_MASK = -1
DOP.DOP$L_SOURCE_INDEX = -1
DOP.DOP$L_FCOLOR = 253
DOP.DOP$L_BCOLOR = 252
DOP.DOP$W_VP_MAX_X = 99
DOP.DOP$W_VP_MAX_Y = 99
DOP.DOP$W_DELTA_X = 0
DOP.DOP$W_DELTA_Y = 0
DOP.DOP$W_VP_MIN_X = 0
DOP.DOP$W_VP_MIN_Y = 0
```

(Continued on next page)

Example 2-5 (Cont.) Deleting a Viewport

```
! Load the Stop values
DOP.DOP$W_ITEM_TYPE = DOP$C_STOP
DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_DRIVER_VP_ID = VIEWPORT_ID

RETURN
END

! *********************
! * F_POLY SUBROUTINE *
! *********************

SUBROUTINE F_POLY (DOP, DOP_VAR, SIZE)
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Associate the predefined variable structure w/ DOP_VAR
RECORD /DOP_POLY_ARRAY/ DOP_VAR

! Load the Common block
DOP.DOP$W_SIZE = SIZE
DOP.DOP$W_FLAGS = 0
DOP.DOP$W_MODE = WRIT$M_NO_SRC_COMP + 10
DOP.DOP$L_MASK = -1
DOP.DOP$L_SOURCE_INDEX = -1
DOP.DOP$L_FCOLOR = 252
DOP.DOP$L_BCOLOR = 252

! Load the POLYGON values
DOP.DOP$W_ITEM_TYPE = DOP$C_FILL_POLYGON
DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_BITMAP_ID = 0 ! no bitmap

DOP_VAR.DOP_POLY$W_LEFT_X1 = 10
DOP_VAR.DOP_POLY$W_LEFT_Y1 = 10
DOP_VAR.DOP_POLY$W_LEFT_X2 = 10
DOP_VAR.DOP_POLY$W_LEFT_Y2 = 110

DOP_VAR.DOP_POLY$W_RIGHT_X1 = 110
DOP_VAR.DOP_POLY$W_RIGHT_Y1 = 10
DOP_VAR.DOP_POLY$W_RIGHT_X2 = 110
DOP_VAR.DOP_POLY$W_RIGHT_Y2 = 110

RETURN
END
```

2.17 Moving a Viewport

The QDSS driver does not provide support to move a viewport or change its size. However, an application can define a move operation.

To move a viewport, an application must perform the following steps:

1. Copy the contents of the old viewport to an area in the offscreen bitmap.
2. Delete the old viewport.
3. Create a new viewport.
4. Copy the data from the offscreen bitmap to the new viewport.

How to perform all of these functions is shown in the previous examples.

2.18 Using the Deferred Queue

When a viewport is occluded and the free area of offscreen memory is already full of occluded regions, the only way to ensure that a drawing operation is performed for the region is to place it on the deferred queue. (Again, keeping track of offscreen memory usage is the responsibility of the application.)

To place a region on the deferred queue:

1. Copy the region to processor memory using the Read Bitmap QIO. This saves the state of the region until it can be updated.
2. Redefine it with the Define Viewport Region QIO setting the absolute base coordinates to (-1,-1). When placing a region on the deferred queue, the relative coordinates are not important. These base coordinates indicate to the driver that operations for the region are to be stored on the deferred queue.

If an application plans to use the deferred queue, it should call the Notify Deferred Queue Full QIO before using the queue. This QIO permits you to notify an application when the deferred queue is full. (It prevents a deferred region from consuming too much memory.)

Your application should execute operations stored on the deferred queue either when QDSS memory becomes available or when notified that the queue is full. To execute an operation on the deferred queue:

1. Copy the region back into offscreen memory using the Write Bitmap QIO.
2. Redefine the region using the Define Viewport Region QIO.
3. Execute operations stored on the queue using the Execute Deferred Queue QIO.

If no memory is available when the queue is full, swap another region out of offscreen memory and onto the deferred queue, at least until the queue is executed.

Note that if a viewport is occluded in more than one place, you may have to execute *the same deferred queue — multiple times.* That is, update the first region at one point, and update the other region with the same operations later. To do this, define a region on the deferred queue when you redefine the first region to be in offscreen memory for deferred queue execution. This will inform the driver that the viewport still has an occluded region and will prevent it from deleting the deferred queue.

After the deferred queue drawing operations have been executed to all the deferred regions of a viewport, delete the drawing operations from the deferred queue with the Delete Deferred Queue Operation QIO. Also, when an application deletes a viewport with a region on the deferred queue, delete the deferred queue drawing operations for that region.

2.19 Using Color

The QDSS driver uses several planes of memory to display color. Corresponding points in each plane of memory are mapped to a single pixel on the display. The number of memory planes configured with a system determines the depth or Z-mode of a pixel and the number of colors that can be simultaneously displayed.

A driver configured with n planes of memory is capable of displaying 2**n colors. The QDSS driver can be configured with four or eight planes of memory. Hence, a four-plane system can display 16 simultaneous colors and an eight-plane system 256.

The total number of different colors a system could display is specified by a longword in the QDSS QVB block. The QDSS driver is capable of defining a maximum of 2**24 colors, but only 16 or 236 colors may be on the screen at any one time.

Each color that can be displayed is represented by a value in the *hardware color map* (also known as the hardware look-up table). On color systems, a color is defined by three 16-bit intensity values, one for each primary color. On intensity systems, a color is represented by only one 16-bit value. The low-order eight bits of these values are ignored. The high-order eight bits are used to represent the actual intensity values. Hence, intensity values range from 0 to 255.

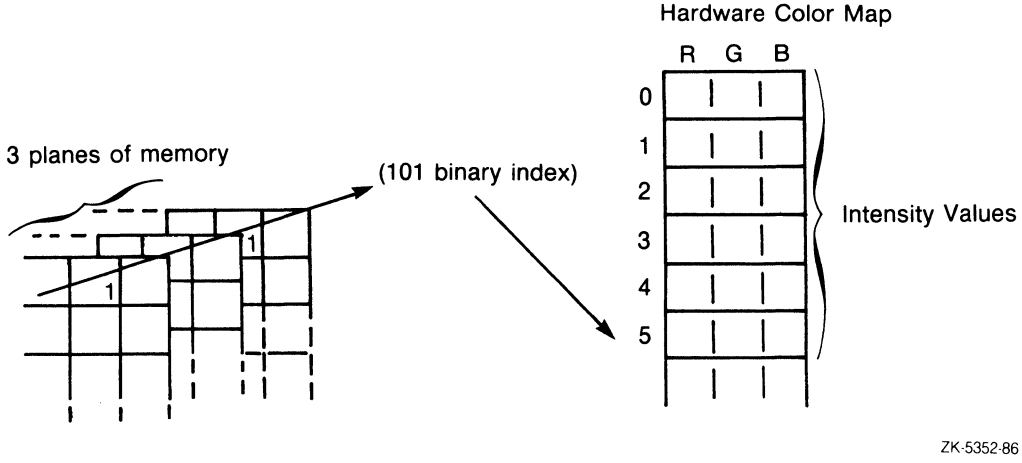
2.19.1 Informing the Driver About Color

Before an application can use the hardware color map, it must tell the driver which type of color system it is using. To identify a system as either color or intensity based, use the Set Color Characteristics QIO, specifying the second unique parameter as either 0 (indicating a color system) or 1 (indicating an intensity system). Once this has been determined, the driver will only accept Set Color Map Entries QIO requests that match this setting. All other Set Color Map Entries QIOs are rejected.

2.19.2 Manipulating Color Map Values

The values used to represent pixels onscreen are used as indexes into the hardware color map. Pixel values are read in the Z-mode direction — for example, if only the first three planes of a pixel are used and the bits in the first and third planes are set, the resulting pixel value is 101 (binary). This value indexes into the fifth value in the hardware color map. Figure 2–11 illustrates how this works.

Figure 2-11 Indexing the Hardware Color Map



If your application is not using the UIS environment, it must load any values it uses into the hardware color map. To load values into the hardware color map, use the Set Color Map Entries QIO; specifying an index into the color map at which to begin initializing, the address of the buffer containing the desired intensity values, and the length of the buffer. You may call this QIO at any time to redefine the values in the color map.

To determine the current values of the hardware color map, use the Get Color Map Entries QIO; specifying an index into the color map at which to begin retrieving information, the address of the buffer to hold the returned intensity values, and the length of the buffer.

Chapter 3

QVSS/QDSS Common QIO Interface

This chapter contains descriptions of the QIO calls that you can use with the QVSS and QDSS drivers. The QIO descriptions appear in alphabetical order. The following list organizes the QIOs in functional groups.

Functional Group	QIO Name
Controlling the Keyboard	Enable Keyboard Input Enable Keyboard Sound Modify Keyboard Characteristics
Controlling Input	Enable Input Simulation Get Next Input Token
Controlling the Pointer	Define Pointer Cursor Pattern Enable Button Transition Enable Pointer Motion
Controlling the Screen	Initialize Screen Modify Systemwide Characteristics Enable Function Keys
Controlling the Tablet	Enable Data Digitizing
Controlling User Entry Lists	Enable User Entry
Obtaining Information	Get Keyboard Characteristics Get Number of List Entries Get System Information
Using Compose Keys	Load Compose Sequence Table Revert to Default Compose Table
Using Soft Keys	Load Keyboard Table Revert to Default Keyboard Table

3.1 How to Use This Chapter

Before attempting to call the QIOs described in this chapter, you should read Chapters 1 and 2 of this manual, and familiarize yourself with Appendixes A, B, and H of this manual. Chapters 1 and 2 describe the general operation of the QIOs. Appendixes A and B contain pictures and descriptions of the data types that you pass to the driver through the **P1** to **P6** parameters of several of the QIOs. Appendix H contains a complete description of the SYS$QIO system service.

3.1.1 QIO Description Format

The QIO descriptions contained in this chapter follow a strict format. You should use them as reference material when you are coding a specific QIO.

The following list notes the main headings in each QIO description and describes the type of information that appears there.

- **QIO Name** — Name of the QIO described. Appears in each description.
- **Overview** — A brief description of the operation the QIO performs. Appears in each description.
- **Format** — The format of the call to SYS$QIO. This format line contains the QIO function code you must pass to SYS$QIO to perform the desired operation.

  Arguments in brackets are optional arguments. In some programming languages, such as MACRO, you do not have to specify optional arguments; when you omit them, the assembler supplies a default value of 0. Some other programming languages, such as FORTRAN, do *not* allow you to omit optional arguments; you must pass a value of 0 for any argument you do not want to specify. Check the documentation for your programming language to see how it handles optional arguments. Appears in each description.
- **Unique Parameters** — List of the information that you must pass to the driver through $QIO's **P1** to **P6** parameters. Each parameter description also tells you whether the parameter is required or optional. Appears in each description.
- **Description** — Additional information about the operation of the QIO call. Optional.
- **Example** — Example of the QIO. Optional.

Define Pointer Cursor Pattern

Defines the pointer cursor pattern for a given region on the physical screen. When the cursor enters that region, the new cursor pattern takes effect. Other arguments allow the user to select the cursor style and reposition the pointer cursor.

Format

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

Unique Parameters

P1 — IO$C_QV_SETCURSOR

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_SETCURSOR function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QV_BIND	Binds the pointer to the region specified in **P6**. Once the pointer enters the specified region, it cannot move outside the region's borders. If the region becomes occluded, the pointer is no longer bound to the region.
IO$M_QV_DELETE	Deletes the specified pointer cursor pattern request. Any data contained in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.
IO$M_QV_LAST	Places the specified pointer cursor pattern request last in the entry list. If IO$M_QV_LAST is not specified, the request is placed first in the list. If an outstanding pointer cursor pattern request exists for the channel, it is updated to reflect the new entry.

Function Modifier	Action
IO$M_QV_LOAD_DEFAULT	Makes the specified cursor pointer the system default cursor pointer. If you specify this function modifier, the system ignores any screen region you specify in **P6**.
IO$M_QV_USE_DEFAULT	Requests that the system use the system default cursor pointer when the region specified in **P6** becomes active. If you specify this function modifier, the system ignores any arguments you specify in **P2**, **P4**, and **P5**.
IO$M_QV_TWO_PLANE_ CURSOR	Indicates the system is loading a multiplane cursor pattern. Use this modifier to load a cursor pattern on a QDSS system. Refer to the Description section for more information on multiplane cursors.

P2 — address of bitmap image

This parameter is either a 16-word array or, on multiplane cursor systems, a 32-word array. The QVB contains a field that allows you to determine whether yours is a single-plane or multiplane system. Use the Get System Information QIO to access this field. (See Description section.)

This is an optional parameter.

P3 — address of new cursor position

This parameter is a longword that points to a two-longword array that defines the new cursor position. The first longword specifies the X coordinate of the new cursor position in pixels. The second longword specifies the Y coordinate of the new cursor position in pixels. The following diagram shows the data structure that defines the new cursor position.

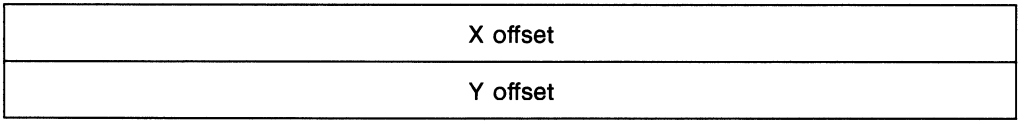
X position on physical screen
Y position on physical screen

Field	Use
X position on physical screen	Specifies X coordinate in pixels
Y position on physical screen	Specifies Y coordinate in pixels

If P3 is 0, the pointer cursor is not repositioned.

P4 — address of pointer cursor hot spot definition

This parameter is an address that points to a two-longword array that defines the pointer cursor hot spot, which is that point within the 16- x 16-pixel cursor display region that is the actual cursor position. The following diagram shows the data structure that defines the cursor hot spot.



Field	Use
X offset	The X offset in pixels from the upper left corner of the pointer pattern to the active point.
Y offset	The Y offset in pixels from the upper left corner of the pointer pattern to the active point.

P5 — value that defines cursor style

This parameter is a longword value in the range 0 through 3 that defines how the cursor is presented against the background screen. The following table lists each value and the style it denotes.

Value	Style
0	Dynamic NAND. The background under the cursor "hot spot" is examined and, if it is black (all off), the cursor is NANDed with the background. If the background is not black, the cursor is ORed with the background.
1	Dynamic OR. The background under the cursor "hot spot" is examined and, if it is black (all off), the cursor is ORed with the background. If the background is not black, the cursor is NANDed with the background.
2	NAND. The cursor is always NANDed with the background screen.
3	OR. The cursor is always ORed with the background screen.

This parameter is ignored for multiplane cursor systems.

P6 — address of screen rectangle values block

This parameter is a longword that points to a screen rectangle values block. This block defines a rectangle on the screen. The following diagram shows the data structure that defines the screen rectangle.

MINX (left side value)
MINY (bottom side value)
MAXX (right side value)
MAXY (top side value)

Field	Use
MINX (left side value)	Pixel value for left side of rectangle
MINY (bottom side value)	Pixel value for bottom side of rectangle
MAXX (right side value)	Pixel value for right side of rectangle
MAXY (top side value)	Pixel value for top side of rectangle

If you do not specify P6, a default rectangle that covers the entire screen is used.

Description

When the pointer cursor moves outside a currently active rectangle, a special signal notifies the process that the cursor has left the region.

The QVSS and QDSS drivers allow you to specify a pointer cursor pattern that defines the shape of the cursor (QDSS systems use a multiplane cursor which is described in the following section). The shape can be in the form of a block, a cross, an arrow, or any other desired configuration. You can also define the cursor style (how the cursor is presented against the background screen) and the location of the cursor hot spot (the point within the cursor pattern region that is the actual cursor position). In addition to moving the cursor with the pointer, you can also reposition the cursor by specifying new X and Y cursor coordinates.

Multiplane Cursor Patterns

If your system uses a multiplane cursor (QDSS), you may specify a 32-word array as a cursor pattern. Currently, multiplane cursors consist of two planes. Typically, you use two planes to prevent the cursor from disappearing when it is moved over varying backgrounds. To understand how the two planes work, think of the 32-word array as two 16-word arrays, array A and array B.

The bit pattern in array A is determined in the following way:

- A 1 in a bit indicates that the corresponding pixel be filled.

- A 0 in a bit indicates that whatever is on the screen at the corresponding pixel should show through (remember, the cursor is overlaid on the screen).

The bit pattern in array B uses the the bits set to 0 in array A as a mask; those corresponding bits are ignored in array B. The remaining bit pattern in array B is determined in the following way:

- A 1 in a bit indicates that the corresponding pixel be filled with the background color.
- A 0 in a bit indicates that the corresponding pixel be filled with the foreground color.

Example

The example that follows shows the typical assignment of a pointer cursor region.

```
.
.
.
QV$CURSOR1:                                    ; INITIAL 16 X 16 CURSOR
                                               ; PATTERN
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
.
.
.
```

Define Pointer Cursor Pattern

```
REGION1:                                    ; CURSOR REGION 1
        .LONG   20
        .LONG   20
        .LONG   300
        .LONG   300
    .
    .
    .
20$:    MOVL    #IO$C_QV_SETCURSOR,R0       ; DEFINE CURSOR 1
        $QIOW_S CHAN=CUR_CHAN1,-            ; ASSIGNED CHANNEL
                FUNC=#IO$_SETMODE,-         ; SET MODE QIO
                P1 =(R0),-                  ; CURSOR PATTERN REQUEST
                P2=#QV$CURSOR1,-            ; CURSOR DESCRIPTION
                P6=#REGION1                 ; CURSOR REGION
        BLBS    R0,30$                      ; NO ERROR IF SET
        BRW     ERROR

30$:    MOVL    #IO$C_QV_SETCURSOR,R0       ; DEFINE CURSOR 2
        $QIOW_S CHAN=CUR_CHAN2,-            ; SECOND ASSIGNED CHANNEL
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#QV$CURSOR2,-
                P6=#REGION2
        BLBS    R0,40$                      ; NO ERROR IF SET
        BRW     ERROR
    .
    .
    .
```

Enable Button Transition

Enables repeating pointer button ASTs for the process on the specified channel. If this request has the highest priority for the specified rectangle, each button transition delivers an AST when the pointer cursor enters that area of the physical screen.

Format

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

Unique Parameters

P1 — IO$C_QV_ENABUTTON

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_ENABUTTON function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QV_DELETE	Deletes the specified pointer button request. Any data contained in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.
IO$M_QV_LAST	Places the specified pointer button request last in the list. If IO$M_QV_LAST is not specified, the request is placed first in the list. If an outstanding pointer button request exists for the channel, it is updated to reflect the new priority.
IO$M_QV_PURG_TAH	Purges the type-ahead buffer of any existing pointer button transitions.

P2 — address of a pointer button AST specification block

This parameter is a longword that points to a pointer AST specification block. This block specifies a user-supplied AST routine that is notified each time a pointer button transition occurs.

The following diagram shows the data structure that specifies a pointer button AST.

Enable Button Transition

AST service routine address
AST parameter
access mode
input token address

Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. If no AST routine is specified, input is stored in the type-ahead buffer, and delivered either when an AST region is declared, or when a Get Next Input Token QIO is issued. The type-ahead buffer holds 32 input tokens or characters.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.

Field	Use
input token address	The input token address is the address of a longword that receives an input token when an AST routine is called. Word 0 of the longword receives token or character data. The token is a decimal value indicating which button is activated. The values are assigned to the pointer buttons sequentially, starting with the select button, which is always 400. The driver stores the token in the low-order word of the longword. Bit 15 of the high-order word determines whether the transition is up or down; 1 equals down, 0 equals up. The rest of the high-order word contains more control information that can be used to determine if the Shift, Ctrl, or Lock keys are depressed. You can use these keys as meta-keys (keys used in combination). Bit 14 corresponds to the Shift key, bit 13 corresponds to the Ctrl key, and bit 12 corresponds to the Lock key. When the bit is set, the key is down.

This parameter must be specified.

P3 — must be 0

P4 — address of a pointer button characteristics block

This parameter is a longword that points to a pointer button characteristics block. This block specifies which button-related characteristics to enable or disable for the button region. When the region becomes active, the specified characteristics become active.

Enable Button Transition

The following diagram shows the data structure that specifies pointer button characteristics.

enabled characteristics mask
disabled characteristics mask
0
0

Field	Use
enabled characteristics mask	Longword of characteristics to be enabled.
disabled characteristics mask	Longword of characteristics to be disabled.
	The pointer button characteristics, which are defined by the $QVBDEF macro, consist of the following bit:

Characteristic	Meaning
QV$M_BUT_ UPTODOWN	After a pointer button down transition occurs, the current pointer button request receives all future pointer button transitions until all pointer buttons return to the up position (regardless of the position of the pointer cursor on the physical screen). If this characteristic is disabled, then each up and down transition is delivered to whichever button request is active for the current pointer cursor position. Default is on.

0	This longword must be zero.
0	This longword must be zero.

P5 — must be 0

P6 — address of a screen rectangle values block

This parameter is a longword that points to a screen rectangle values block. This block defines the area on the physical screen for which the specified button transition is enabled.

The following diagram shows the data structure that defines the screen rectangle.

MINX (left side value)
MINY (bottom side value)
MAXX (right side value)
MAXY (top side value)

Field	Use
MINX (left side value)	Pixel value for left side of rectangle
MINY (bottom side value)	Pixel value for bottom side of rectangle
MAXX (right side value)	Pixel value for right side of rectangle
MAXY (top side value)	Pixel value for top side of rectangle

This parameter is optional. If you do not specify a screen rectangle values block, a default rectangle that covers the entire screen is used.

Description

The QVSS and QDSS drivers support a multibutton pointer. The drivers allow a process to enable a *pointer button request* to signal the occurrence of a *pointer button transition,* either up or down. A token is passed to the specified AST routine to signal which button made a transition, and the type of transition (up or down). Many applications are only interested in pointer button events that occur in a specific region of the physical screen. The **P6** parameter of the pointer button request specifies a rectangle on the physical screen. This rectangle defines the area where the application is interested in pointer button transitions. If rectangles for pointer button requests for multiple channels (or processes) overlap, priority is given to the first rectangle on the list.

Example

The example that follows shows the typical programming and use of pointer button ASTs.

```
    .
    .
    .
SET_BUTTONAST:
        $ASSIGN_S   DEVNAM=WS_DEVNAM,- ; ASSIGN CHANNEL USING
                    CHAN=BUT_CHAN     ;  LOGICAL NAME AND
                                      ;  CHANNEL NUMBER
        BLBS    R0,10$                ; NO ERROR IF SET
        BRW     ERROR                 ; ERROR
10$:    MOVL    #IO$C_QV_ENABUTTON,R0
        $QIOW_S CHAN=BUT_CHAN,-
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#BUT_BLOCK,-
                P6=#BUT_REGION
        BLBS    R0,20$                ; NO ERROR IF SET
        BRW     ERROR
20$:    RSB
BUT_BLOCK:                            ; BUTTON AST
                                      ; SPECIFICATION BLOCK
        .LONG   BUT_AST               ; AST ADDRESS
        .LONG   0                     ; AST PARAMETER
        .LONG   0                     ; ACCESS MODE
        .LONG   BUTTON                ; BUTTON INFORMATION
                                      ; LONGWORD
BUT_REGION:                           ; AST REGION
        .LONG   20
        .LONG   20
        .LONG   300
        .LONG   300
    .
    .
    .
```

Enable Data Digitizing

If the system pointing device is a tablet, the Enable Data Digitizing QIO allows you to use the tablet as a data digitizer.

Format

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

Unique Parameters

P1 — IO$C_QV_ENABLE_DIGITIZING

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_ENABLE_DIGITIZING function code with the following optional function modifier:

Function Modifier	Action
IO$M_QV_DELETE	Deletes the specified data digitizing request. Any data contained in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.

P2 — address of a pointer movement AST specification block

This parameter is a longword that points to a pointer movement AST specification block. This block specifies a user-supplied AST routine that is notified when pointer movement occurs inside the data rectangle you specify in **P6**. The pointer position is reported using the best granularity in which the device can report.

The following diagram shows the data structure that specifies a pointer movement AST for the tablet.

AST service routine address
AST parameter
access mode
address of new pointer cursor position

Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. No buffering of data in the type-ahead buffer occurs for pointer motion ASTs.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST is maximized with the current access mode.
address of new pointer cursor position	The fourth longword contains the address of a longword to receive the new pointer cursor position when the AST routine is called. (If your application does not need this information, specify a 0.) The low-order word receives the new X pixel location of the pointer cursor; the high-order word receives the new Y pixel location of the cursor. The range of X is defined in the *qvb$w_tablet_width* field of the QVB block; the range of Y is defined in the *qvb$w_tablet_height* field of the QVB block.

This is an optional parameter.

P3 — address of a pointer button AST specification block

This parameter is a longword that points to a pointer button AST specification block. This block specifies a user-supplied AST that is notified when a button transition occurs.

The following diagram shows the data structure that specifies the pointer button AST for the tablet.

AST service routine address
AST parameter
access mode
input token address

Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. If no AST routine is specified, input is stored in the type-ahead buffer, and delivered either when an AST region is declared, or when a Get Next Input Token QIO is issued. The type-ahead buffer holds 32 input tokens or characters.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.

Field	Use
input token address	The input token address is the address of a longword that receives an input token when an AST routine is called. Word 0 of the longword receives token or character data. The token is a decimal value indicating which button was activated. The values are assigned to the pointer buttons sequentially starting with the select button, which is always 400. The driver stores the token in the low-order word of the longword. Bit 15 of the high-order word determines whether the transition is up or down; 1 equals down, 0 equals up. The rest of the high-order word contains more control information that can be used to determine if the Shift, Ctrl, or Lock keys are depressed. You can use these keys as meta-keys (keys used in combination). Bit 14 corresponds to the Shift key, bit 13 corresponds to the Ctrl key, and bit 12 corresponds to the Lock key. When the bit is set, the key is down.

This is an optional parameter.

P4, P5 — must be 0

P6 — address of a data rectangle values block

This parameter is a longword that points to a data rectangle values block. This block defines the active rectangle on the tablet. The origin (0,0) of the tablet is the lower left-hand corner.

The following diagram shows the data structure that defines a data rectangle.

MINX (left side value)
MINY (bottom side value)
MAXX (right side value)
MAXY (top side value)

Field	Use
MINX (left side value)	Pixel value for left side of rectangle
MINY (bottom side value)	Pixel value for bottom side of rectangle
MAXX (right side value)	Pixel value for right side of rectangle
MAXY (top side value)	Pixel value for top side of rectangle

This parameter is optional. If you do not specify a data rectangle values block, a default rectangle that covers the entire tablet is used.

Description

Only the process that issues the data digitizing request may change or cancel it. Upon process deletion, any outstanding data digitizing is canceled.

Only one data digitizing region may be active at a time. When one process has declared a data digitizing region, attempts by other processes to declare an additional data digitizing region fail.

Enable Function Keys

Allows the windowing system to access function keys F1 through F5, which are reserved for workstation control functions and should not be used in application programs. These keys are defined by the driver, which, in addition to informing the owner of the key of the keypress, performs special functions.

Format

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

Unique Parameters

P1 — IO$C_QV_ENAFNKEY

The function code that identifies the action that the QIO performs.

This parameter must be specified.

P2 — address of a reserved function keystroke AST specification block

This parameter is a longword that points to a reserved function keystroke AST specification block. This block specifies a user-supplied AST routine that is notified each time a keystroke occurs.

The following diagram shows the data structure that specifies a reserved function keystroke AST.

AST service routine address
AST parameter
access mode
input token address

Field	Use
AST service routine address	The address of the AST service routine. Specify 0 if no AST routine is required. If no AST routine is specified, input is stored in the type-ahead buffer, and delivered either when an AST region is declared, or when a Get Next Input Token QIO is issued. The type-ahead buffer holds 32 input tokens or characters.
AST parameter	The user-defined AST parameter delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.
input token address	The address of a longword that receives an input token when an AST routine is called; word 0 of the longword contains token data defined by the $SMGDEF macro for these function keys. By default, an AST is only signaled on a down transition.

This parameter must be specified.

P3 — symbolic name for function key to associate with request

This parameter is a symbolic name that indicates the function key to associate with this request. The following bits are defined:

Key Value	Function
QV$M_KEY_F1	Driver signals AST and toggles keyboard Hold Screen lamp.
QV$M_KEY_F2	Operator screen. If the SYSGEN parameter WS_OPA0 is set to 1, toggles between the workstation screen and the operator screen.
QV$M_KEY_F3	Switch window. If an alternate windowing system is enabled, the driver signals an AST and toggles between the windowing systems. (This applies to monochrome VAXstation I and II workstations.)
QV$M_KEY_F4	The driver signals an AST.
QV$M_KEY_F5	The driver signals an AST.

P4, P5, P6 — must be 0

Example

The following example shows typical programming for the F5 function key.

```
    .
    .
    .
10$:    MOVL    #IO$C_QV_ENAFNKEY,R0     ; FUNC KEY REQUEST TO R0
        $QIOW_S CHAN=FNKEY_F5_CHAN,-     ; ASSIGNED CHANNEL
                FUNC=#IO$_SETMODE,-      ; SET MODE QIO
                P1=(R0),-                ; FUNCTION KEY REQUEST
                P2=#FNKEY_BLOCK,-        ; AST SPEC BLOCK
                P3=#QV$M_KEY_F5          ; KEY IS F5
        BLBS    R0,30$                   ; NO ERROR IF SET
        BRW     ERROR
    .
    .
    .
FNKEY_BLOCK:                             ; FUNCTION KEY AST
                                         ; SPECIFICATION BLOCK
        .LONG   F5_AST                   ; AST ADDRESS
        .LONG   F5_ACK                   ; AST PARAMETER
        .LONG   0                        ; ACCESS MODE
        .LONG   CHARACTER                ; INPUT TOKEN STORAGE
    .
    .
    .
```

Enable Input Simulation

Simulates keystrokes, pointer motion, and pointer button transitions.

Format

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

Unique Parameters

P1 — IO$C_QV_SIMULATE

The function code that identifies what action the QIO performs.

This parameter must be specified.

P2 — address of ASCII text descriptor

This parameter is a longword that points to a descriptor for the ASCII text to send to the current keyboard region. The maximum number of characters allowed in the text string is 32. If P2 is 0, no data is sent.

This is an optional parameter.

P3 — address of new pointer position

This parameter is a longword that points to a two-longword array that defines a new pointer position. The first longword specifies the X coordinate of the new pointer position in pixels; the second longword specifies the Y coordinate.

The following diagram shows the data structure that specifies the new pointer position.

X position on physical screen
Y position on physical screen

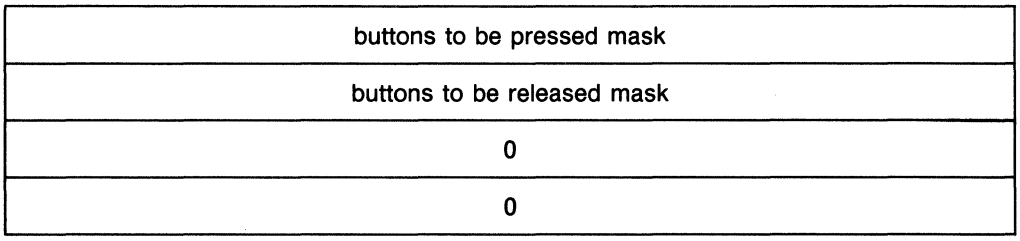
Field	Use
X position on the physical screen	Specifies X coordinate in pixels
Y position on the physical screen	Specifies Y coordinate in pixels

This is an optional parameter. If P3 is 0, the pointer is not repositioned.

P4 — address of button simulation block

This parameter is a longword that points to a button simulation block that specifies which pointer buttons are pressed or released.

The following diagram shows a button simulation block.



Field	Use
Buttons to be pressed mask	Mask of the buttons to be pressed
Buttons to be released mask	Mask of the buttons to be released
	The pointer button definitions used in the masks are defined by the $QVBDEF macro. They consist of the following symbols:

Symbol	Meaning
QV$M_BUTTON_1	Select button
QV$M_BUTTON_2	Button 2
QV$M_BUTTON_3	Button 3
QV$M_BUTTON_4	Button 4

0	This longword must be zero.
0	This longword must be zero.

This is an optional parameter. If P4 is zero, the pointer buttons are not modified.

P5, P6 — must be 0

Example

The example that follows shows typical programming for input simulation.

```
    .
    .
    .
5$:     BSBW    SET_CHARACTERISTICS     ; SET UP SYSTEM
                                        ;  CHARACTERISTICS
        BSBW    SET_PERM_CURSOR         ; SET UP NEW SYSTEMWIDE
                                        ;  CURSOR PATTERN
    .
    .
    .
        BSBW    SET_MOUSEAST            ; SET UP MOUSE REGION AST
        BSBW    SIMULATE_INPUT          ; SIMULATE INPUT ON
                                        ;  KEYBOARD 2.
        $CLREF_S        EFN=#2          ; CLEAR EVENT FLAG #2
        $WAITFR_S       EFN=#2          ; WAIT FOR EVENT FLAG #2
ERROR:  $EXIT_S R0
    .
    .
    .
SIMULATE_INPUT:
        MOVL    #IO$C_QV_SIMULATE,R0    ; SIMULATE KEYBOARD INPUT
        $QIOW_S CHAN=KBD_CHAN2,-        ;  ON KEYBOARD CHANNEL 2
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#SIM_ACK,-
                P3=#0
        BLBS    R0,20$                  ; NO ERROR IF SET
        BRW     ERROR
20$:    RSB
SIM_ACK:
        .ASCID  /This input SIMULATED on chan 2./
    .
    .
    .
```

Enable Keyboard Input

Enables repeating character input ASTs for the process on the specified channel.

Format

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

Unique Parameters

P1 — IO$C_QV_ENAKB

The function code that identifies the action that the QIO performs. This parameter must be specified.

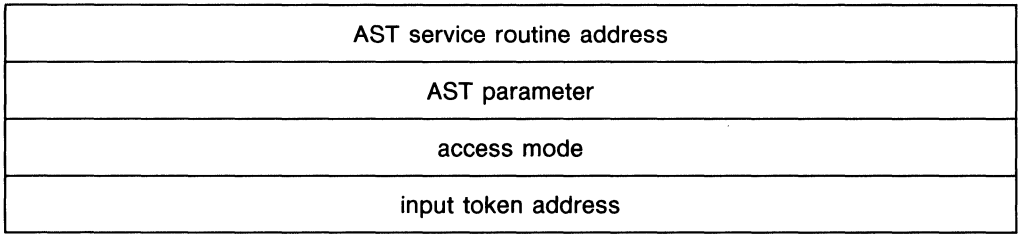
To modify the action of the QIO, you can OR the IO$C_QV_ENAKB function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QV_CYCLE	Removes the active keyboard from the head of the keyboard request list and places it at the end of the list (lowest priority). The next highest priority keyboard request then becomes the active keyboard request and a control AST is delivered on its behalf.
IO$M_QV_DELETE	Deletes the specified keyboard request. Any data contained in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.
IO$M_QV_LAST	Places the specified keyboard request last in the list. If IO$M_QV_LAST is not specified, the request is placed first in the list. If an outstanding keyboard request exists for the channel, it is updated to reflect the new priority.
IO$M_QV_PURG_TAH	Purges the type-ahead buffer of any keyboard request on this channel.

P2 — address of a keystroke AST specification block

This parameter is a longword address that points to a keystroke AST specification block. This block specifies a user-supplied AST routine that is notified each time a keystroke occurs.

The following diagram shows the data structure that specifies a keystroke AST.



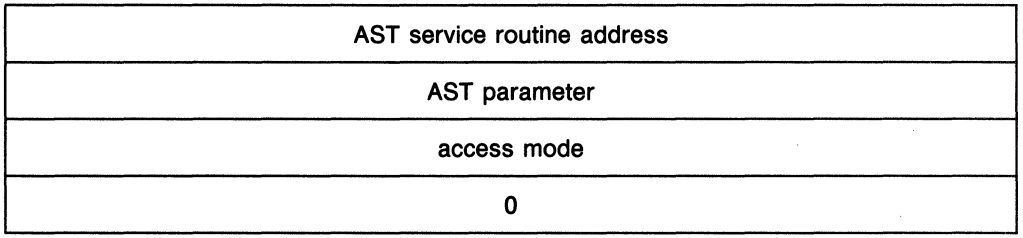
Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. If no AST routine is specified, input is stored in the type-ahead buffer, and delivered either when an AST region is declared, or when a Get Next Input Token QIO is issued.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.
input token address	The input token address is the address of a longword that receives an input token when an AST routine is called. Word 0 of the longword contains token or character data. Values in the range of 0 to 255 map into the DIGITAL multinational character set. Values in the range of 256 to 512 map function keys into token values. Word 1 of the longword contains control information; bit 15 defines the status of a token (1 equals down, 0 equals up). By default an AST is only signaled on a down transition. The rest of the high-order word contains more control information that can be used to determine if the Shift , Ctrl , or Lock keys are depressed. You can use these keys as meta-keys (keys used in combination). Bit 14 corresponds to the Shift key, bit 13 corresponds to the Ctrl key, and bit 12 corresponds to the Lock key. When the bit is set, the key is down.

This parameter must be specified.

P3 — address of a keyboard request AST specification block

This parameter is a longword address that points to a keyboard request AST specification block. This block specifies a control AST routine that is notified when a keyboard request becomes active. A keyboard request becomes active when the active keyboard owner is deleted or a cycle request causes it to become active. No control AST is delivered when the new request is already active or the owning process issued the cycle request.

The following diagram shows the data structure that specifies a keyboard request AST.



Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. If no AST routine is specified, input is stored in the type-ahead buffer, and delivered either when an AST region is declared, or when a Get Next Input Token QIO is issued. The type-ahead buffer holds 32 input tokens or characters.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.
0	The fourth longword must be zero.

This is an optional parameter.

P4 — address of a keyboard characteristics block

This parameter is a longword address that points to a keyboard characteristics block. The keyboard characteristics block describes keyboard-related characteristics that are to be enabled or disabled for the keyboard region. The specified characteristics are enabled or disabled when the keyboard region becomes active.

The keyboard characteristics block is ignored if the keyboard region for this channel already exists. To modify the characteristics of an existing keyboard region, use the Modify Keyboard Characteristics QIO.

The default characteristics are those specified in the systemwide characteristics block, which can be modified using the Modify Systemwide Characteristics QIO. The current systemwide characteristics are stored in the characteristics field of the QVB.

The following diagram shows the data structure that specifies the keyboard characteristics block.

enabled characteristics mask
disabled characteristics mask
keyclick volume
0

Enable Keyboard Input

Field	Use
enabled characteristics mask	The first longword is a mask of characteristics to be enabled.
disabled characteristics mask	The second longword is a mask of characteristics to be disabled.
	The keyboard characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_KEY_AUTORPT	Key held down automatically repeats. Default is on.
QV$M_KEY_KEYCLICK	Keyclick sounds on each keystroke. Default is on.
QV$M_KEY_UDF6	Function keys F6 through F10 generate up/down transitions. Default is off.
QV$M_KEY_UDF11	Function keys F11 through F14 generate up/down transitions. Default is off.

Field	Use	
	QV$M_KEY_UDF17	Function keys F17 through F20 generate up/down transitions. Default is off.
	QV$M_KEY_UDHELPDO	Function keys HELP and DO generate up/down transitions. Default is off.
	QV$M_KEY_UDE1	Function keys E1 through E6 generate up/down transitions. Default is off.
	QV$M_KEY_UDARROW	Arrow keys generate up/down transitions. Default is off.
	QV$M_KEY_UDNUMKEY	Numeric keypad keys generate up/down transitions. Default is off.
keyclick volume	The keyclick volume must be a value in the range of 1 to 8; 1 is loudest and 8 is softest. If a value of 0 is specified, the current system default keyclick volume is used.	
0	The fourth longword must be 0.	

This is an optional parameter.

P5, P6 — must be 0

Example

The example that follow shows a typical assignment of two terminal channels, keyboard requests on those channels, and associated AST routines.

```
    .
    .
    .
P2_BLOCK1:                          ; AST SPECIFICATION BLOCK 1
        .LONG   KBD_AST             ; AST ADDRESS
        .LONG   ACK1                ; AST PARAMETER
        .LONG   0                   ; AST DELIVERY MODE
        .LONG   CHARACTER           ; INPUT TOKEN
ACK1:   .ASCID  /INPUT ACKNOWLEDGED CHANNEL 1/

P2_BLOCK2:                          ; AST SPECIFICATION BLOCK 2
        .LONG   KBD_AST
        .LONG   ACK2
        .LONG   0
        .LONG   CHARACTER
ACK2:   .ASCID  /INPUT ACKNOWLEDGED CHANNEL 2/

P3_BLOCK:                           ; CONTROL AST SPECIFICATION
                                    ; BLOCK
        .LONG   CTL_AST             ; CONTROL AST ADDRESS
        .LONG   0                   ; AST PARAMETER
        .LONG   0                   ; AST DELIVERY MODE
        .LONG   0                   ; MUST BE ZERO
    .
    .
    .
SET_KBDAST:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,- ; ASSIGN CHANNEL USING
                   CHAN=KBD_CHAN2   ; LOGICAL NAME AND
                                    ;  CHANNEL NUMBER
        BLBS    R0,5$               ; NO ERROR IF SET
        BRW     ERROR               ; ERROR
    .
    .
    .
20$:    MOVL    #IO$C_QV_ENAKB,R0   ; ENABLE KEYBOARD AST
                                    ; REQUEST TO R0
        $QIOW_S CHAN=KBD_CHAN2,-    ; ASSIGNED CHANNEL
                FUNC=#IO$_SETMODE,- ; SET MODE QIO
                P1=(R0),-           ; KEYBOARD AST REQUEST
                P2=#P2_BLOCK2,-     ; USER AST ROUTINE
                P3=#P3_BLOCK        ; CONTROL AST ROUTINE
        BLBS    R0,30$              ; NO ERROR IF SET
```

```
        BRW     ERROR
    .
    .
    .
KBD_AST:
F5_AST:
        .WORD
        PUSHL   4(AP)                   ; SEND ACKNOWLEDGMENT
        CALLS   #1,G^LIB$PUT_LINE       ; MESSAGE
        BLBS    R0, 10$
5$:     BRW     ERROR

10$:    CMPW    #KEY$C_F5,CHARACTER     ; WAS F5 TYPED?
        BNEQ    20$
        BSBW    CYCLE_KBD               ; CYCLE THE KEYBOARD LIST
        BRB     40$                     ;  AND EXIT

20$:    PUSHAL  DESC                    ; SEND CHARACTER TYPED
        CALLS   #1,G^LIB$PUT_LINE
        BLBC    R0,5$

        CMPB    #^A/C/,CHARACTER        ; WAS A "C" TYPED?
        BNEQ    30$
        BSBW    CYCLE_KBD               ; CYCLE THE KEYBOARD LIST
        BRB     40$

30$:    CMPB    #^A/F/,CHARACTER        ; WAS AN "F" TYPED?
        BNEQ    40$
        $SETEF_S EFN=#2                 ; YES, EXIT PROGRAM

40$:    RET
    .
    .
    .
CTL_AST:
        .WORD
        PUSHAL   CYCLE                  ; SEND ACKNOWLEDGMENT
        CALLS    #1,G^LIB$PUT_LINE      ; MESSAGE
        BLBS     R0, 10$
5$:     BRW      ERROR

10$:    RET
    .
    .
    .
```

Enable Keyboard Sound

Enables a process to make a bell or keyclick sound on the LK201 keyboard.

Format

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

Unique Parameters

P1 — IO$C_QV_SOUND

The function code that identifies the action that the QIO performs.

This parameter must be specified.

P2 — symbolic name that denotes type of sound

This parameter is a symbolic name that denotes the type of sound. The sound types, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_SOUND_BELL	Sound bell.
QV$M_SOUND_CLICK	Sound keyclick.

This parameter must be specified.

P3 — value that specifies the sound volume

This parameter specifies the sound volume, which must be a value in the range of 1 to 8; 1 is loudest and 8 is softest. If a value of 0 is indicated, the previously specified (that is, current) volume is used.

This is an optional parameter.

P4, P5, P6 — must be 0

Example

The following example shows how the bell sound can be programmed.

```
.
.
.
20$:    MOVL    #IO$C_QV_SOUND,R0       ; SOUND REQUEST TO R0
        $QIOW_S CHAN=SYS_CHAN1,-         ; ASSIGNED CHANNEL
                FUNC=#IO$_SETMODE,-      ; SET MODE QIO
                P1 = (R0),-              ; SOUND REQUEST
                P2=#QV$M_SOUND_BELL      ; SOUND TYPE IS BELL
        BLBS    R0,20$                   ; NO ERROR IF SET
        BRW     ERROR
.
.
.
```

Enable Pointer Movement

Enables repeating pointer motion ASTs for the process on the specified channel. If this request has the highest priority for the specified rectangle, each pointer motion delivers an AST when the pointer cursor enters the specified area of the physical screen.

Format

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

Unique Parameters

P1 — IO$C_QV_MOUSEMOV

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_MOUSEMOV function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QV_DELETE	Deletes the specified pointer motion request. Any data contained in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.
IO$M_QV_LAST	Places the specified pointer motion request last in the list. If IO$M_QV_LAST is not specified, the request is placed first in the list. If an outstanding pointer motion request exists for the channel, it is updated to reflect the new priority.

P2 — address of a pointer motion AST specification block

This parameter is a longword that points to a pointer motion AST specification block. This block specifies a user-supplied AST routine that is notified when pointer motion occurs.

The following diagram shows the data structure that specifies a pointer motion AST.

Enable Pointer Movement

AST service routine address
AST parameter
access mode
address of new pointer cursor position

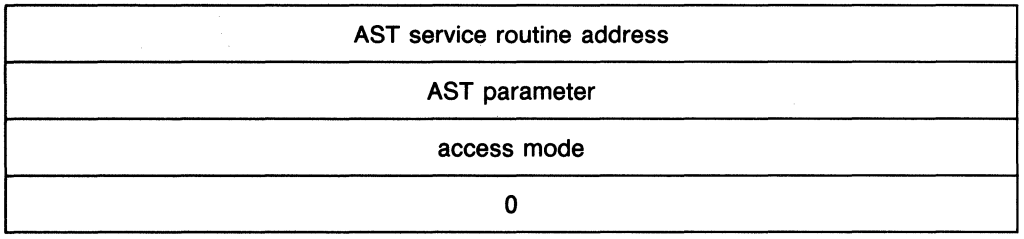
Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. No buffering of data in the type-ahead buffer occurs for pointer motion ASTs.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.
address of new pointer cursor position	The fourth longword contains the address of a longword to receive the new pointer cursor position when the AST routine is called. (If your application does not need this information, specify a 0.) The low-order word receives the new X pixel location of the pointer cursor; the high-order word receives the new Y pixel location of the cursor. For screen pointers, X is in the range 0 through 1023 with the lowest value denoting the left side of the screen; Y is in the range 0 through 863 with the lowest value denoting the bottom of the screen. For tablet pointers, the range of X is defined in the *qvb$w_tablet_width* field of the QVSS block; the range of Y is defined in the *qvb$w_tablet_height* field of the QVB block.

This parameter must be specified.

P3 — address of a pointer cursor exit AST specification block

This parameter is a longword that points to a pointer cursor exit AST specification block. This block specifies a user-supplied control AST routine that is notified when the pointer cursor exits from the rectangle specified by **P6**.

The following diagram show the data structure that specifies a pointer cursor exit AST.



Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST is maximized with the current access mode.
0	The fourth longword must be zero.

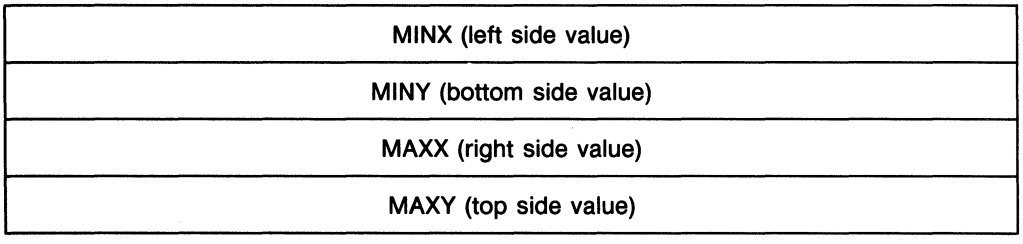
This is an optional parameter.

P4, P5 — must be 0

P6 — address of a screen rectangle values block

This parameter is a longword that points to a screen rectangle values block. This block defines a rectangle on the screen.

The following diagram shows the data structure that specifies a screen rectangle.



Field	Use
MINX (left side value)	Pixel value for left side of rectangle
MINY (bottom side value)	Pixel value for bottom side of rectangle
MAXX (right side value)	Pixel value for right side of rectangle
MAXY (top side value)	Pixel value for top side of rectangle

This is an optional parameter. If you do not specify P6, a default rectangle that covers the entire screen is used.

Description

The QVSS and QDSS drivers track the pointer by moving the *pointer cursor* on the physical screen. In order to minimize desktop space required to manipulate the pointer, the driver updates the pointer cursor on the screen proportionally to the velocity at which the pointer is being moved on the desktop. The driver allows a process to enable a pointer motion notification request to signal pointer motion within a selected area of the physical screen. A token is passed to the specified AST address to indicate the new pointer cursor physical position. An input rectangle defines the area in which the application is interested in pointer motion. If rectangles for pointer motion requests for multiple channels (or processes) overlap, priority is given to the first rectangle on the list.

Example

The following example shows how a pointer motion AST could be programmed.

```
    .
    .
    .
10$:    MOVL    #IO$C_QV_MOUSEMOV,R0     ; ENABLE MOUSE MOTION
        $QIOW_S CHAN=MOUSE_CHAN,-        ; REGION
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#MOUSE_BLOCK,-
                P6=#MOUSE_REGION
        BLBS    R0,20$                   ; NO ERROR IF SET
        BRW     ERROR
20$:    RSB
MOUSE_BLOCK:                             ; MOUSE REGION AST
                                         ; SPECIFICATION BLOCK
        .LONG   MOUSE_AST                ; AST ADDRESS
        .LONG   MOUSE_ACK                ; AST PARAMETER
        .LONG   0                        ; ACCESS MODE
        .LONG   MOUSE_XY                 ; NEW MOUSE CURSOR POSITION
                                         ; STORAGE

MOUSE_REGION:                            ; MOUSE REGION
        .LONG   400
        .LONG   400
        .LONG   800
        .LONG   800
    .
    .
    .
```

Enable User Entry

Assigns a control AST to each user entry in an optional graphics package entry list. The entry placed at the top of the list receives a control AST when a cycle request occurs, that is, an entry control AST request that includes the IO$M_QV_CYCLE function modifier.

Format

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

Unique Parameters

P1 — IO$C_QV_ENAUSER

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_ENABUSER function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QV_CYCLE	Removes the active entry from the head of the entry list and places it at the end of the list (lowest priority). The next highest priority keyboard request then becomes the active keyboard request and a control AST is delivered on its behalf.
IO$M_QV_DELETE	Deletes the specified entry control request. Any data contained in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.
IO$M_QV_LAST	Places the specified entry control request last in the list. If IO$M_QV_LAST is not specified, the request is placed first in the list. If an outstanding entry control request exists for the channel, it is updated to reflect the new priority.

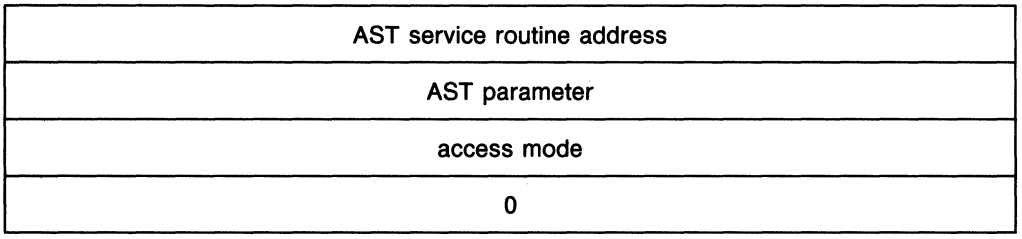
P2 — must be 0

This parameter must be specified.

P3 — address of an active entry AST specification block

This parameter is a longword that points to an active entry AST specification block. This block specifies a user-supplied control AST routine that is notified when this entry becomes active.

The following diagram shows the data structure that specifies an active entry AST.



Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.
0	The fourth longword must be 0.

This parameter must be specified.

P4, P5, P6 — must be 0

Get Keyboard Characteristics

Obtains the keyboard characteristics for keyboard regions that already exist. The specified keyboard region is not changed and does not become the active keyboard region.

Format

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

Unique Parameters

P1 — IO$C_QV_GETKB_INFO

The function code that identifies the action that the QIO performs.

This parameter must be specified.

P2, P3 — must be 0

P4 — address of a keyboard characteristics block

This parameter is a longword that points to a keyboard characteristics block. This block receives the keyboard-related characteristics for this keyboard region.

The following diagram shows the data structure into which the driver returns the keyboard characteristics.

enabled characteristics mask
disabled characteristics mask
keyclick volume
0

Get Keyboard Characteristics

Field	Use
enabled characteristics mask	The first longword is a mask of characteristics which are enabled.
disabled characteristics mask	The second longword is a mask of characteristics which are disabled.
	The keyboard characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_KEY_AUTORPT	Key held down automatically repeats. Default is on.
QV$M_KEY_KEYCLICK	Keyclick sounds on each keystroke. Default is on.
QV$M_KEY_UDF6	Function keys F6 through F10 generate up/down transitions. Default is off.
QV$M_KEY_UDF11	Function keys F11 through F14 generate up/down transitions. Default is off.
QV$M_KEY_UDF17	Function keys F17 through F20 generate up/down transitions. Default is off.
QV$M_KEY_ UDHELPDO	Function keys HELP and DO generate up/down transitions. Default is off.
QV$M_KEY_UDE1	Function keys E1 through E6 generate up/down transitions. Default is off.
QV$M_KEY_ UDARROW	Arrow keys generate up/down transitions. Default is off.
QV$M_KEY_ UDNUMKEY	Numeric keypad keys generate up/down transitions. Default is off.

Field	Use
keyclick volume	The keyclick volume is a value in the range of 1 to 8; 1 is loudest and 8 is softest. If a value of 0 is returned, the current system default keyclick volume is used.
0	The fourth longword must be 0.

This parameter must be specified.

P5, P6 — must be 0

Get Next Input Token

Intercepts the next input event for an entry on a list and returns it in the second longword of the I/O status block.

Format

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

Unique Parameters

P1 — must be 0

This parameter must be specified.

P2 — symbolic name of list from which token is intercepted

This parameter specifies the list on which the get function is performed. One of the following lists must be specified:

List	Function Performed
IO$C_QV_ENABUTTON	Get next pointer button change.
IO$C_QV_ENAKB	Get next token entered from the keyboard.
IO$C_QV_MOUSEMOV	Get next pointer motion.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

If the type-ahead buffer for that entry is empty, the QIO is held and then delivered as soon as the next token is obtained. If the type-ahead buffer is not empty, the first character in the type-ahead buffer is removed and delivered. Only one token is returned for each QIO issued. Regarding the decision as to where to deliver the next input token, this QIO preempts the AST routine for the entry. It is possible to queue several Get Next Input Token QIOs on a single entry.

This QIO cannot be issued if the entry has an AST routine currently enabled. However, an AST routine can be enabled even though one or more of these QIOs may be outstanding on the entry.

Get Number of List Entries

Allows you to obtain the number of AST entries contained in any of the following entry lists:

- Keyboard entry list
- Pointer button transition list
- Pointer motion list
- User entry list

Format

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

Unique Parameters

P1 — IO$C_QV_GET_ENTRIES

This parameter must be specified.

P2 — symbolic name of list from which you want to get entry count

This parameter specifies the list on which the get function is performed. One of the following lists must be specified:

List	Function Performed
IO$C_QV_ENAKB	Get number of entries in the keyboard entry list
IO$C_QV_ENABUTTON	Get number of entries in the pointer button transition list
IO$C_QV_MOUSEMOV	Get number of entries in the pointer motion list
IO$C_QV_ENAUSER	Get number of entries in the user entry list

This parameter must be specified.

P3 — address of a longword to receive the number of entries

This parameter must be specified.

P4, P5, P6 — must be 0

Get System Information

Returns the address of the system information block, which may be used by a process to obtain dynamic video-related information. The address of the system information block does not change. Therefore, a process need only obtain the address once and can reference it until the process terminates.

Format

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

Unique Parameters

P1 — IO$C_QV_GETSYS

The function code that identifies the action that the QIO performs.

This parameter must be specified.

P2 — address of quadword block

This parameter is a longword that points to a quadword block to receive the address and length of the system information block. The structure is user read, kernel read/write.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The system information block differs depending on whether you are using a QVSS system or a QDSS system. The system information block of the QVSS system is referred to as the QVB. The system information block of the QDSS system is referred to as the QDB. Both structures contain fields that hold information about video memory. Some of the information is static, such as the address of onscreen memory; other information is dynamic, such as the current pointer position.

Appendix A contains a full illustration and explanation of the QVB. Appendix B contains a full illustration and explanation of the QDB.

Initialize Screen

Initializes the screen to a known state.

Format

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

Unique Parameters

P1 — IO$C_QV_INITIALIZE function code

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2, P3, P4, P5, P6 — must be 0

Description

The Initialize Screen QIO initializes the workstation screen. You must initialize the screen whenever you initialize a windowing system that is going to use the QVSS or QDSS screen. The windowing system should issue this QIO only once, before it issues any other QIOs to the driver.

Load Compose Sequence Table

Loads two-stroke and three-stroke compose sequence tables.

Format

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

Unique Parameters

P1 — IO$C_QV_LOAD_COMPOSE_TABLE

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_LOAD_COMPOSE_TABLE function code with the following optional function modifier:

Function Modifier	Action
IO$M_QV_LOAD_DEFAULT	If the IO$M_QV_LOAD_DEFAULT modifier is specified, this table is the default table for the entire workstation. You should specify this modifier only once when you load the first table after you initialize the workstation.

P2 — size of a two-stroke compose sequence table

This parameter is a longword that contains the size (in bytes) of the two-stroke compose sequence table.

This is an optional parameter.

P3 — address of two-stroke compose sequence table

This parameter is a longword that points to the two-stroke compose sequence table. For a complete description of the two-stroke compose sequence table, see Chapter 2.

This is an optional parameter.

P4 — size of three-stroke compose sequence table

This parameter is a longword that contains the size (in bytes) of the three-stroke compose sequence table.

This is an optional parameter.

P5 — address of the three–stroke compose sequence table

This parameter is a longword that points to the three–stroke compose sequence table. For a complete description of the three–stroke compose sequence table, see Chapter 2.

This is an optional parameter.

P6 — must be 0

Description

If only one table is to be loaded, specify values of 0 for the parameters of the other table.

A keyboard region has two tables associated with it for each type of compose sequence: the default table (taken from the workstation default), and a private table (if one has been loaded).

Example

The following example shows how to load a three-stroke compose sequence table.

Load Compose Sequence Table

```
    .
    .
    .
SET_COMPOSE3_TABLE:
        MOVL    #<IO$C_QV_LOAD_COMPOSE_TABLE>, R0
        $QIOW_S CHAN = KBD_CHAN1, -    ; CHANGE THE COMPOSE TABLE
                FUNC = #IO$_SETMODE, -
                P1 = (R0), -
                P4 = #COMPOSE3_TBL_LEN, - ; three-stroke TABLE SIZE
                P5 = #COMPOSE3_TBL        ; three-stroke TABLE ADDR
        BLBS    R0,5$                     ; NOT SET ON ERROR
        BRW     ERROR
5$:     RSB
        VC$COMPOSE_KEYINIT  COMPOSE3_TBL  ; GENERATE AN
                                          ; EMPTY TABLE
                                          ; FILL THE TABLE HERE
                                          ;
        VC$COMPOSE_KEY  <^a/A/>,<^a/"/>,,<^xc4>
        VC$COMPOSE_KEY  <^a/A/>,<^a/'/>,,<^xc1>
        VC$COMPOSE_KEY  <^a/A/>,<^a/*/>,,<^xc5>
        VC$COMPOSE_KEY  <^a/A/>,<^a/A/>,<@>
        VC$COMPOSE_KEY  <^a/A/>,<^a/E/>,,<^xc6> ; ORDER SENSITIVE
        VC$COMPOSE_KEY  <^a/A/>,<^a/^/>,,<^xc2>
        VC$COMPOSE_KEY  <^a/A/>,<^a/_/>,,<^xaa>
    .
    .
    .
        VC$COMPOSE_KEYEND  COMPOSE3_TBL_LEN  ; END THE TABLE
                                             ; AND DETERMINE ITS
                                             ; LENGTH
    .
    .
    .
```

Load Keyboard Table

Loads a keyboard table.

Format

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

Unique Parameters

P1 — IO$C_QV_LOAD_KEY_TABLE

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_LOAD_KEY_TABLE function code with the following optional function modifier:

Function Modifier	Action
IO$M_QV_LOAD_DEFAULT	If the IO$M_QV_LOAD_DEFAULT modifier is specified, this table is the default table for the entire workstation. You should specify this modifier only once when you load the first table after you initialize the workstation.

P2 — size of keyboard table

This parameter is a longword that contains the size (in bytes) of the keyboard table in bytes.

This parameter must be specified.

P3 — address of the keyboard table

This parameter is a longword that points to the keyboard table. For a complete description of keyboard tables, see Chapter 2.

This parameter must be specified.

P4, P5, P6 — must be 0

Description

Each window has two associated tables: the default table (taken from the workstation default), and a private table (if one has been loaded).

Example

The following example shows how to load a keyboard table.

```
    .
    .
    .
SET_FRENCH_KB:
        MOVL    #<IO$C_QV_LOAD_KEY_TABLE>, R0
        $QIOW_S CHAN = KBD_CHAN1, -          ; CHANGE THE KEYBOARD
                FUNC = #IO$_SETMODE, -       ; LAYOUT
                P1 = (R0), -
                P2 = #KB_LAYOUT_TBL_LEN, - ; KEYBOARD TABLE SIZE
                P3 = #KB_LAYOUT_TBL          ; KEYBOARD TABLE
                                             ; ADDRESS
        BLBS    R0,5$                        ; NO ERROR IF SET
        BRW     ERROR
5$:     RSB
    .
    .
    .
```

Modify Keyboard Characteristics

Changes the keyboard characteristics for an existing keyboard region.

Format

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

Unique Parameters

P1 — IO$C_QV_MODIFYKB

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_MODIFYKB function code with the following optional function modifier:

Function Modifier	Action
IO$M_QV_ACTIVE	Removes the active keyboard from the head of the keyboard request list and places it at the end of the list (lowest priority). The keyboard region just modified becomes the active keyboard request and a control AST is delivered on its behalf.

P2 — address of a keystroke AST specification block

This parameter is a longword that points to a keystroke AST specification block. This block specifies a user-supplied AST routine that is notified each time a keystroke occurs.

The following diagram shows the data structure that specifies a keystroke AST.

AST service routine address
AST parameter
access mode
input token address

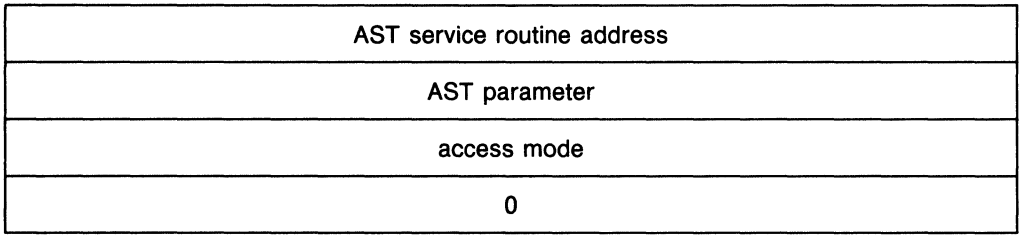
Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. If no AST routine is specified, input is stored in the type-ahead buffer, and delivered either when an AST region is declared, or when a Get Next Input Token QIO is issued. The type-ahead buffer holds 32 input tokens or characters.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it.
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.
input token address	The input token address is the address of a longword that receives an input token when an AST routine is called. Word 0 of the longword contains token or character data. Values in the range of 0 to 255 map into the DIGITAL multinational character set. Values in the range of 256 to 512 map function keys into token values. Word 1 of the longword contains control information; bit 15 defines the status of a token (1 equals down, 0 equals up). By default an AST is only signaled on a down transition.

This is an optional parameter.

P3 — address of a keyboard request AST specification block

This parameter is a longword address that points to a keyboard request AST specification block. This block specifies a control AST routine that is notified when a keyboard request becomes active. A keyboard request becomes active when the active keyboard owner is deleted or a cycle request causes the keyboard request to become active. No control AST is delivered when the new request is already active or the owning process issued the cycle request.

The following diagram shows the data structure that specifies a keyboard request AST.



Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST routine is required. If no AST routine is specified, input is stored in the type-ahead buffer, and delivered either when an AST region is declared, or when a Get Next Input Token QIO is issued. The type-ahead buffer holds 32 input tokens or characters.
AST parameter	The user-defined AST parameter is delivered to the AST routine. The driver does not examine it .
access mode	The access mode at which to deliver the AST. It is maximized with the current access mode.
0	The fourth longword must be zero.

This is an optional parameter.

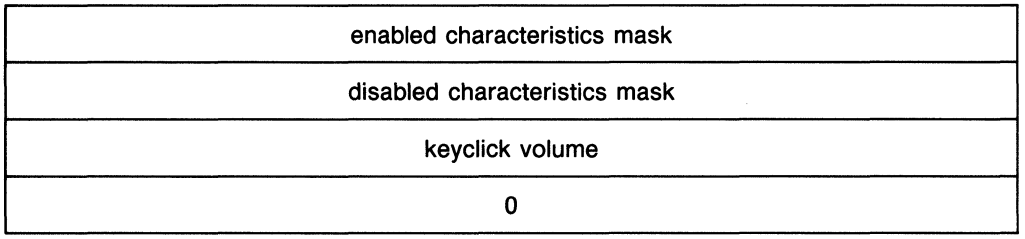
P4 — address of a keyboard characteristics block

This parameter is a longword address that points to a keyboard characteristics block. The keyboard characteristics block describes keyboard-related characteristics that are to be enabled or disabled for the keyboard region. The specified characteristics are enabled or disabled when the keyboard region becomes active.

The default characteristics are those specified in the systemwide characteristics block, which can be modified using the Modify Systemwide Characteristics QIO. The current systemwide characteristics are stored in the characteristics field of the QVB.

Modify Keyboard Characteristics

The following diagram shows the data structure that specifies the keyboard characteristics.



Field	Use
enabled characteristics mask	The first longword is a mask of characteristics to enable.
disabled characteristics mask	The second longword is a mask of characteristics to disable.
	The keyboard characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_KEY_AUTORPT	Key held down automatically repeats. Default is on.
QV$M_KEY_KEYCLICK	Keyclick sounds on each keystroke. Default is on.
QV$M_KEY_UDF6	Function keys F6 through F10 generate up/down transitions. Default is off.
QV$M_KEY_UDF11	Function keys F11 through F14 generate up/down transitions. Default is off.
QV$M_KEY_UDF17	Function keys F17 through F20 generate up/down transitions. Default is off.

Field	Use	
	QV$M_KEY_ UDHELPDO	Function keys HELP and DO generate up/down transitions. Default is off.
	QV$M_KEY_UDE1	Function keys E1 through E6 generate up/down transitions. Default is off.
	QV$M_KEY_ UDARROW	Arrow keys generate up/down transitions. Default is off.
	QV$M_KEY_ UDNUMKEY	Numeric keypad keys generate up/down transitions. Default is off.
keyclick volume	The keyclick volume must be a value in the range of 1 to 8; 1 is loudest and 8 is softest. If a value of 0 is specified, the current system default keyclick volume is used.	
0	The fourth longword must be 0.	

This is an optional parameter.

P5, P6 — must be 0

Modify Systemwide Characteristics

Changes the systemwide windowing characteristics.

Format

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

Unique Parameters

P1 — IO$C_QV_MODIFYSYS

The function code that identifies the action that the QIO performs.

This parameter must be specified.

P2, P3 — must be 0

P4 — address of a system characteristics block

This parameter is a longword address that points to a system characteristics block. This block specifies the system-related characteristics that you want to enable or disable.

The following diagram shows the data structure that specifies system characteristics.

enabled characteristics mask
disabled characteristics mask
keyclick volume
screen saver timeout value

Field	Use
enabled characteristics mask	The first longword is a mask of characteristics to enable.
disabled characteristics mask	The second longword is a mask of characteristics to disable.
	The system characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_KEY_AUTORPT	Key held down automatically repeats. Default is on.
QV$M_KEY_KEYCLICK	Keyclick sounds on each keystroke. Default is on.
QV$M_KEY_UDF6	Function keys F6 through F10 generate up/down transitions. Default is off.
QV$M_KEY_UDF11	Function keys F11 through F14 generate up/down transitions. Default is off.
QV$M_KEY_UDF17	Function keys F17 through F20 generate up/down transitions. Default is off.
QV$M_KEY_UDHELPDO	Function keys HELP and DO generate up/down transitions. Default is off.

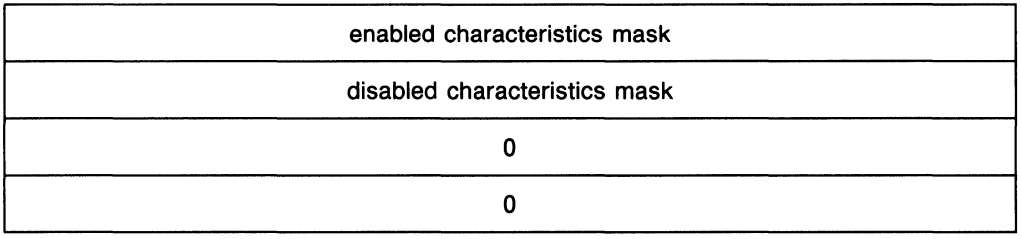
Field	Use	
	QV$M_KEY_UDE1	Function keys E1 through E6 generate up/down transitions. Default is off.
	QV$M_KEY_UDARROW	Arrow keys generate up/down transitions. Default is off.
	QV$M_KEY_UDNUMKEY	Numeric keypad keys generate up/down transitions. Default is off.
	QV$M_SYS_SCRSAV	Disable video output to monitor if no input activity occurs within the number of minutes specified in the fourth longword. Any keystroke, pointer button transition, or pointer motion resets the timer and reactivates a disabled screen. Default is on.
keyclick volume	The keyclick volume must be a value in the range of 1 to 8; 1 is loudest and 8 is softest. Default is 3.	
screen saver timeout value	The screen saver timeout value represents the number of minutes of inactivity that must elapse before the screen saver is activated. The value must be in the range of 1 to 1440. If a value of 0 is specified, then the timeout value is not changed. Default is 15.	

This is an optional parameter.

P5 — address of a pointer characteristics block

This parameter is a longword address that points to a pointer characteristics block. This block specifies the pointer-related characteristics that you want to enable or disable.

The following diagram shows the data structure that specifies pointer characteristics.



Field	Use
enabled characteristics mask	The first longword is a mask of characteristics to be enabled.
disabled characteristics mask	The second longword is a mask of characteristics to be disabled.
	The pointer characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_PTR_LEFT_HAND	Invert buttons on mouse or puck. (Buttons 1 and 3 are switched.)
QV$M_PTR_INVERT_STYLUS	Invert buttons on stylus. (Buttons 1 and 3 are switched.)

0	Must be 0
0	Must be 0

This is an optional parameter.

P6 — must be 0

Example

The following example shows how the system windowing characteristics could be changed.

```
    .
    .
    .
SET_CHARACTERISTICS:
        MOVL    #IO$C_QV_MODIFYSYS,R0 ; CHANGE SYSTEM
        $QIOW_S CHAN=SYS_CHAN1,-      ; CHARACTERISTICS
                FUNC=#IO$_SETMODE,-
                P1 = (R0),-
                P4 = #CHAR_BLOCK
        BLBS    R0,10$                ; NO ERROR IF SET
        BRW     ERROR
    .
    .
    .
CHAR_BLOCK:                           ; CHARACTERISTICS BLOCK
        .LONG   <QV$M_SYS_AUTORPT!QV$M_SYS_KEYCLICK> ; ENABLE
                                      ; THESE CHARACTERISTICS
        .LONG   <QV$M_SYS_UDF6!QV$M_SYS_UDARROW> ; DISABLE
                                      ; THESE CHARACTERISTICS
        .LONG   5                     ; KEYCLICK VOLUME
        .LONG   30                    ; SCREEN SAVER TIMEOUT
    .
    .
    .
```

Revert to Default Compose Table

Reverts to the two-stroke or three-stroke default compose table.

Format

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

Unique Parameters

P1 — IO$C_QV_USE_DEFAULT_TABLE

The function code that identifies the action that the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QV_USE_DEFAULT_TABLE function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QV_COMPOSE2	Reverts to the two-stroke default compose table.
IO$M_QV_COMPOSE3	Reverts to the three-stroke default compose table.

P2, P3, P4, P5, P6 — must be 0

Description

This QIO also returns the space used by a private table (if one was loaded) to pool.

Revert to Default Keyboard Table

Reverts to the default keyboard table.

Format

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

Unique Parameters

P1 — IO$C_QV_USE_DEFAULT_TABLE!IO$M_QV_KEYS

The function code that identifies the action that the QIO performs. The exclamation point (!) in the function code above indicates that you must OR the IO$C_QV_USE_DEFAULT_TABLE function code and the IO$M_QV_KEYS function modifier to perform the QIO.

This parameter must be specified.

P2, P3, P4, P5, P6 — must be 0

Description

This QIO also returns the space used by a private table (if one was loaded) to pool.

Revert to Default Keyboard Table

Reverts to the default keyboard table.

Format

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

Unique Parameters

P1 — IO$C_QV_USE_DEFAULT_TABLE ! IO$M_QV_KEYS

The function code that identifies the action that the QIO performs.

This parameter must be specified.

P2, P3, P4, P5, P6 — must be 0

Description

This QIO also returns the space used by a private table (if one was loaded) to pool.

Chapter 4

QDSS-Specific QIO Interface

This chapter contains descriptions of the QIO calls that you can use only with the QDSS driver. The QIO descriptions appear in alphabetical order. The following list groups the QIOs by function.

Functional Group	QIO Name
Controlling Color	Set Color Characteristics Set Color Map Entries
Defining Viewports	Define Viewport Region
Manipulating the DOP Queues	Delete Deferred Queue Execute Deferred Queue Hold Viewport Activity Insert DOP Release Hold Resume Viewport Activity Start Request Queue Stop Request Queue Suspend Occluded Viewport Activity Suspend Viewport Activity
Obtaining Information	Get Color Map Entries Get Free DOPs Get Viewport ID Notify Deferred Queue Full
Transferring Bitmaps	Load Bitmap Read Bitmap Write Bitmap

4.1 How to Use This Chapter

Before attempting to call the QIOs described in this chapter, you should read Chapters 1 and 2 of this manual, and familiarize yourself with Appendixes A, B,

and H of this manual. Chapters 1 and 2 describe the general operation of the QIOs, the data types you pass to the driver through the **P1** to **P6** parameters of several of the QIOs. Appendix H contains a complete description of the $QIO system service.

4.1.1 QIO Description Format

The QIO descriptions contained in this chapter follow a strict format. You should use them as reference material when you are coding a specific type of QIO.

The following list notes the main headings in each QIO description and describes the type of information that appears there.

- **QIO name** — Name of the QIO described. Appears in each description.
- **Overview** — A brief description of the operation the QIO performs. Appears in each description.
- **Format** — The format of the call to SYS$QIO. This format line contains the QIO function code you must pass to SYS$QIO to perform the desired operation.

  Arguments in brackets are optional arguments. In some programming languages, such as MACRO, you do not have to specify optional arguments; when you omit them, the assembler supplies a default value of 0. Some other programming languages, such as FORTRAN, do *not* allow you to omit optional arguments; you must pass a value of 0 for any argument you do not want to specify. Check the documentation for your programming language to see how it handles optional arguments. Appears in each description.
- **Unique Parameters** — List of the information that you must pass to the driver through the **P1** to **P6** parameters of $QIO. Each parameter description also tells you whether the parameter is required or optional. Appears in each description.
- **Description** — Additional information about the operation of the QIO. Optional.
- **Example** — Example of the QIO. Optional.

Define Viewport Region

Creates or changes the update regions that comprise a viewport.

Format

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

Unique Parameters

P1 — IO$C_QD_SET_VIEWPORT_REGIONS

The function code that identifies the action the QIO performs.

P2 — address of the update region definition buffer

This is a longword that points to the buffer that describes the update regions that define the viewport.

The following diagram shows the data structure that specifies each region.

URD$W_Y_MIN	URD$W_X_MIN
URD$W_Y_MAX	URD$W_X_MAX
URD$W_Y_BASE	URD$W_X_BASE

The following list describes the contents of each field in the update region definition block.

Field	Use
URD$W_X_MIN	Viewport-relative X coordinate of the lower left-hand corner of defined region (in pixels)
URD$W_Y_MIN	Viewport-relative Y coordinate of the lower left-hand corner of defined region (in pixels)
URD$W_X_MAX	Viewport-relative X coordinate of the upper right-hand corner of defined region (in pixels)
URD$W_Y_MAX	Viewport-relative Y coordinate of the upper right-hand corner of defined region (in pixels)
URD$W_X_BASE	Absolute X coordinate from lower left-hand corner (in pixels)
URD$W_Y_BASE	Absolute Y coordinate from lower left-hand corner (in pixels)

This parameter must be specified.

P3 — length of the update region definition buffer

Specify the predefined constant URD$C_LENGTH multiplied by the number of update regions.

This parameter must be specified.

P4 — viewport ID

The viewport ID of the viewport you are defining or changing. Use the Get Viewport ID QIO to obtain a unique viewport ID for any new viewport you want to define.

This parameter must be specified.

P5, P6 — must be 0

Description

The Define Viewport Region QIO function creates or modifies viewport update region definitions. The viewport that appears on the screen may be made up of a number of update regions. To access a viewport, you must call this function once. (A viewport must be made up of at least one update region.) If a viewport is made up of a number of regions, you make this call once, specifying an URD buffer (an array or record) that contains all the URDs.

Before you make a Define Viewport Region QIO call, you must make a Stop Request Queue QIO function call.

A viewport management system uses this function to control the screen layout. When a drawing operation is executed, it is performed on each update region specified in the viewport definition.

Delete Deferred Queue Operation

Deletes any entries in the specified deferred queue.

Format

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

Unique Parameters

P1 — IO$C_QD_DELETE_DEFERRED

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — viewport_id

The viewport ID of the viewport associated with the deferred queue. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Delete Deferred Queue Operation QIO function deletes all operations on the deferred queue associated with the specified viewport.

Typically, you use this call when you are popping an occluded viewport. You call it after an Execute Deferred Queue QIO has been executed for every update region previously stored in processor memory.

Execute Deferred Queue

Processes any DOPs on the specified viewport's deferred queue.

Format

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

Unique Parameters

P1 — IO$C_QD_EXECUTE_DEFERRED

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — viewport_id

The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Execute Deferred Queue QIO executes all operations on the deferred queue. All operations executed when the viewport was not accessible are replayed to the specified viewport buffer.

The driver puts operations on the deferred queue when an operation is specified for a screen region that the QDSS hardware cannot currently access. The QDSS hardware cannot access a region when it is stored in processor memory.

Operations on the deferred queue are executed into the currently defined set of region descriptors. If the buffer identifies visible screen locations, some unusual visual effects may be seen.

Get Color Map Entries

Returns the values in the color map.

Format

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

Unique Parameters

P1 — IO$C_QD_GET_COLOR

The function code that identifies the action the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QD_GET_COLOR function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QD_INTENSITY	Specify this value to get a map entry on an intensity system.
IO$M_RESERVED_COLORS	If the IO$M_RESERVED_COLORS modifier is specified, the starting color map entry (P4) is interpreted as the IO$C_QD_TWO_COLOR_CURSOR (set two-color cursor) parameter. In this case, the buffer contains two map entries worth of data: two words for intensity systems, six words for color systems.

P2 — address of a color buffer

This is a longword pointing to a buffer to hold the returned entries of the color map.

The size of the color buffer differs depending on whether you are using a color system or an intensity system. On a color system, the buffer must have an "RGB triple" for each map entry you wish to read. An "RGB triple" is made up of three word-long values, one for red, one for green, and one for blue. So, if you want to read five color map entries, the color buffer must be 15 words long.

On an intensity system, the buffer must have a single word-long value for each map entry you want to read. So, if you want to read five color map entries, the color buffer must be five words long.

Only the eight most significant bits are used for color definition.

This parameter must be specified.

P3 — length of the color buffer, in bytes

On color systems, this must be a multiple of 6. On intensity systems this must be a multiple of 2.

This parameter must be specified.

P4 — starting color map entry

The color map index to use for the first RGB or intensity value specified in the buffer.

This parameter must be specified.

P5, P6 — must be 0

Description

The Get Color Map Entries allows an application to read the color map. Color map information is stored in the QDB. You can access the QDB by issuing the Get System Information QIO. See Appendix B for a full description of the QDB.

Get Free DOPs

Specifies how many DOPs must be on the specified return queue for request queue processing to continue.

Format

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

Unique Parameters

P1 — IO$C_QD_GET_FREE_DOPS

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — number of DOPs

The number of DOPs on the return queue that the driver should wait for before resuming request queue processing.

This parameter must be specified.

P3 — queue flag

Flag specifying which return queue to wait for: a value of 0 indicates the normal (small DOP) queue; a value of 1 indicates the large DOP queue.

This parameter must be specified.

P4 — viewport_id

The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P5, P6 — must be 0

Description

The Get Free DOPs QIO causes a process to wait until there are a specified number of drawing output primitives (DOPs) on the specified viewport return queue.

This function can be used to suspend operations until the driver returns a set of DOPs to the user. This way, the storage space for the DOPs can be reused. Note that you specify the number and size of the DOPs for which to wait.

When this QIO completes and/or the associated AST is fired, the application allocates storage from the specified return queue (with the REMQUE instruction). In some rare instances, this QIO completes and nothing is on the return queue. When this occurs, reissue the QIO until the REMQUE succeeds.

Get Viewport ID

Returns a unique viewport identifier, known as a viewport ID.

Format

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

Unique Parameters

P1 — IO$C_QD_GET_VIEWPORT_ID

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — address of the viewport ID buffer

This is a longword pointing to a longword buffer into which the driver returns the viewport ID.

This parameter must be specified.

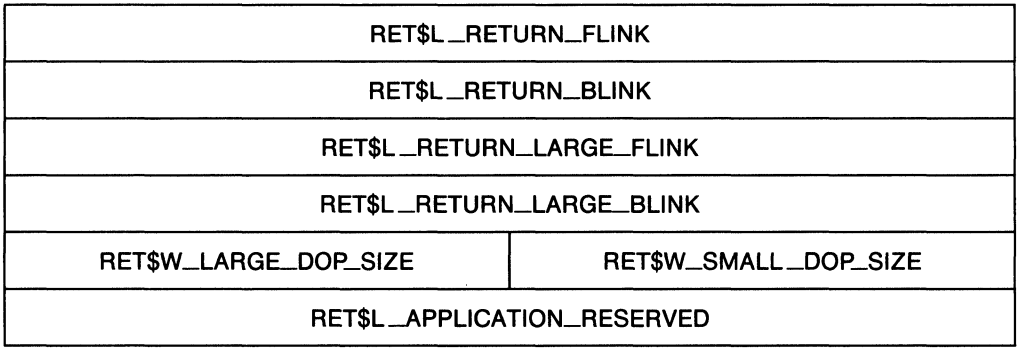
P3 — length of viewport ID buffer, in bytes

The value of this parameter must be 4.

P4 — address of alternate return queue

By default (if this parameter is 0), each viewport is associated with two return queues. This optional parameter permits you to specify an address for an alternate return queue. For example, you may want to share return queues on a per-process basis instead of a per-viewport basis, or you may to want have a systemwide return queue.

The following diagram shows the data structure that specifies an alternate return queue.



The following list describes the contents of each field in the request queue definition.

Field	Use
RET$L_RETURN_FLINK	Forward link for the ordinary return queue
RET$L_RETURN_BLINK	Backward link for the ordinary return queue
RET$L_RETURN_LARGE_FLINK	Forward link for the large DOP return queue
RET$L_RETURN_LARGE_BLINK	Backward link for the large DOP return queue
RET$W_LARGE_DOP_SIZE	Size of a DOP returned to the large DOP return queue
RET$W_SMALL_DOP_SIZE	Size of a DOP returned to the ordinary return queue
RET$L_APPLICATION_RESERVED	A longword reserved for use by the application

P5, P6 — must be 0

Description

The Get Viewport ID QIO returns the unique viewport identifier. This identifier must be supplied as a parameter to several QIO functions. Note that this identifier is also the address of the DOP queue data structure (that contains the request queue and the return queue structures).

You call this function at viewport creation time after you have an assigned channel. It must be called before any function that requires a viewport ID. It may be called only once for each channel; multiple calls result in multiple viewports

Hold Viewport Activity

Stops activity on all viewports except the systemwide viewport.

Format

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

Unique Parameters

P1 — IO$C_QD_HOLD

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2, P3, P4, P5, P6 — must be 0

Description

The Hold Viewport Activity QIO function stops all activity on all viewports except the systemwide viewport.

You use this function when moving a viewport or when implementing a hold screen function.

Insert DOP

Inserts a DOP on the request queue of the specified viewport. Optionally, it notifies the caller of a request queue entry completion.

Format

SYS$QIO *[efn] ,chan*
,IO$_QD_WRITE!IO$M_QD_INSERT_DOP ,[iosb]
,[astadr] ,[astprm] ,p1 ,p2 ,p3 [,p4] [,p5] [,p6]

Unique Parameters

P1 — address of the DOP entry
A longword that points to the DOP entry for which you want completion notification.

This parameter must be specified.

P2 — length of the DOP entry
A longword that contains the length of the DOP that you are inserting into the queue.

This parameter must be specified.

P3 — viewport_id
The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P4, P5, P6 — must be 0

Description

NOTE: The exclamation point (!) in the function code above indicates that you must OR IO$_QD_WRITE and IO$M_QD_INSERT_DOP to perform the Insert DOP QIO.

The Insert DOP QIO permits you to enter a drawing operation primitive (DOP) on the request queue of a specified viewport.

If you specify the IOSB parameter, the system notifies your application when the request queue entry finishes executing.

You can use the Insert DOP to synchronize drawing. If you insert a Stop Viewport Activity DOP on the queue (as opposed to issuing a Stop Request Queue QIO), you can guarantee that the DOPs inserted before the stop are executed.

You should use this QIO only when your application requires notification that the request queue entry is finished executing. An INSQUE instruction or a UISDC$QUEUE_DOP routine is much more efficient.

Load Bitmap

Makes available a bitmap for use by a subsequent text, patterned line, move, or fill operation.

Format

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

Unique Parameters

P1 — IO$C_QD_LOAD_BITMAP

The function code that identifies the action the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QD_LOAD_BITMAP function code with the following optional function modifier:

Function Modifier	Action
IO$M_QD_SYSTEM_WIDE	Defines a bitmap that lasts for the life of the system. (By default, the bitmap is invalid after the channel it is originally returned on is deassigned.)

P2 — address of the bitmap block

A longword pointing to the area of processor memory that describes the desired bitmap. You may also specify a zero in this parameter to postpone dynamic loading of the bitmap — see the Description section for details.

This parameter must be specified.

P3 — length of the bitmap block, in bytes

A longword that contains the length of the bitmap block in bytes. This value must be a multiple of 2.

This parameter must be specified.

P4 — address of a longword for the returned bitmap ID

A longword into which the driver returns the bitmap ID. Subsequent drawing operations reference the loaded bitmap with the bitmap ID. All viewports and processes using the bitmap should refer to the bitmap with this bitmap ID.

This parameter must be specified.

P5 — width of the bitmap, in bits

A longword that contains the width of the bitmap, in bits. The maximum bitmap width you can specify is 1024.

If the bitmap is a single bit per pixel, you must specify a multiple of 16 bits.

This parameter must be specified.

P6 — bits per pixel

A longword that contains the number of bits per pixel. Currently 1 and 8 are supported. Specify the value 1 when a foreground and background color is sufficient (that is, for writing text). Specify the value 8 when you want the full use of color (that is, for natural images).

This parameter must be specified.

Description

The Load Bitmap QIO makes available a bitmap that is used by a drawing operation. Bitmaps may specify the following:

- Fonts
- Line styles
- Fill patterns
- Images

When you load a bitmap, you load it from VAX memory to offscreen memory. Once it is in offscreen memory, drawing operations can access it by using its bitmap ID. You need to load the bitmap only once each time the system is bootstrapped (unless you explicitly delete it with the Delete Bitmap DOP).

Passing the Bitmap

You must pass the bitmap to Load Bitmap in specific form because it is addressed using the X,Y coordinates rather than a single index.

Bitmap storage is defined in blocks of video memory. The size of a bitmap block differs depending on whether you are using a single-plane or a multiplane image.

- Single-plane bitmap blocks are 70 bits by 1024 bits. They can be thought of as 70 lines, each of which is 128 bytes wide.

- Multiplane bitmap blocks are 35 bits by 8180 bits. They can be thought of as 35 lines, each of which is 1024 bytes long. Each consecutive eight bits in a line describes one pixel (in 4-plane systems the high 4 bits are ignored).

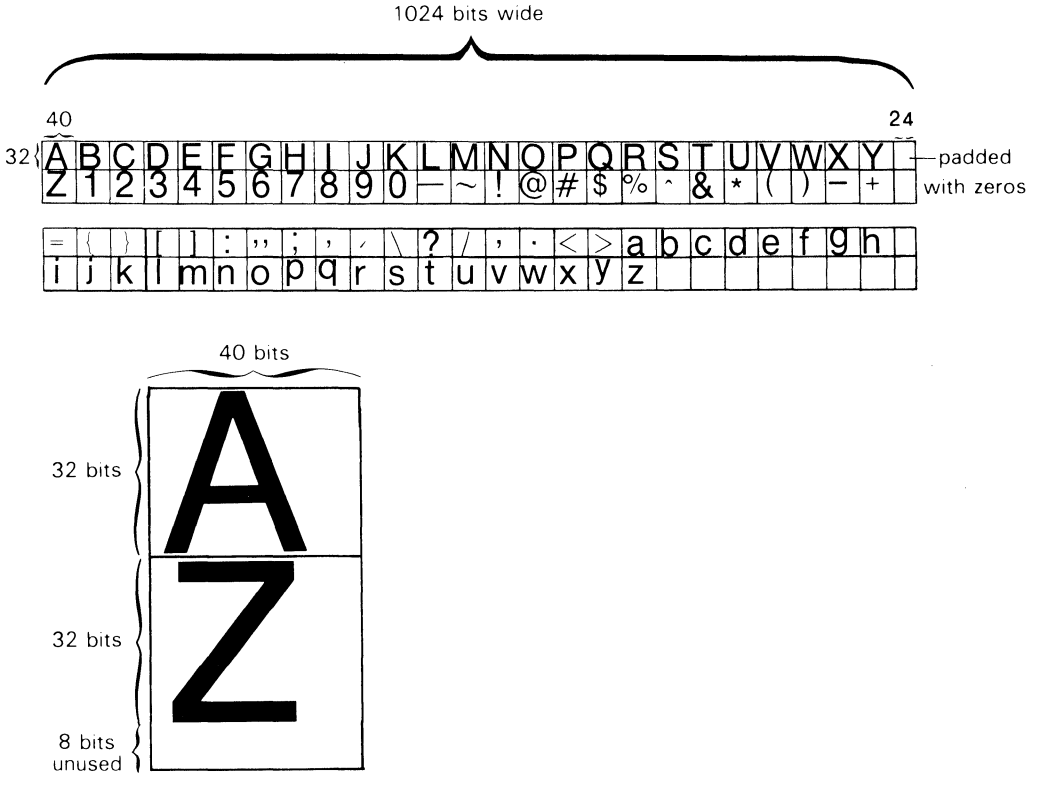
If a bitmap requires more room than can be provided by a single bitmap block, the bitmap must be stored in multiple blocks. This involves getting a separate bitmap ID for each block and keeping track of them in your application.

Bitmaps are passed in blocks to the driver as follows:

1. Each bitmap character (glyph) is stored in a rectangle within the bitmap. Glyphs are stored sequentially and may be packed to the bit; however, it is more efficient to start each character on a byte boundary. (The nature of multiplane bitmaps is such that this occurs naturally.)
2. If the total width of all glyphs exceeds the bitmap block limit, you must use another bitmap block, which you load separately.
3. Any unused portion of a single-plane bitmap line must be padded with zeros to the nearest word boundary. (Again, the nature of multiplane bitmaps is such that this occurs naturally.)

Figure 4–1 following is an example of a large font that uses more than one single-plane bitmap block. Each character is 32 bits high by 40 bits wide. (To include the lowercase letters, you would have to use yet another bitmap block.) Notice that the characters are aligned on word boundaries for better performance. If space is a greater consideration than performance, you could pack the characters to fit more in each bitmap block. Remember, your application must load each block separately and keep track of the different bitmap IDs.

Figure 4-1 Large Font Defined Across Bitmap Blocks



Loading the Bitmap

There are two ways to use Load Bitmap:

- One causes the bitmap to be copied from the user's buffer into a buffer maintained by the driver. When your application accesses the bitmap, it is copied from the driver-maintained buffer into offscreen memory.
- The other creates a handle (identifier) for the bitmap, but relies on the application to supply the bitmap when it is accessed. This second method saves space by not loading a bitmap into offscreen memory until an application accesses it.

To have the driver maintain the bitmap — Specify the address of the bitmap in process memory as the bitmap address parameter. To access the bitmap in a subsequent DOP, load the **bitmap_ID** field of the DOP Common block with the bitmap ID returned by Load Bitmap. The system handles the storage of this bitmap.

When the system manages bitmap storage, it uses the bitmap glyph as a backing store address if it has to swap the bitmap out of offscreen memory. That is, when the bitmap is accessed, the system uses the address in the bitmap glyph to swap the bitmap back into offscreen memory.

To dynamically load a bitmap — Specify zero as the bitmap address parameter, but still specify the correct length, width, and bits-per-pixel. To access the bitmap in a subsequent DOP, load the **bitmap_ID** field of the DOP Common block with the bitmap ID returned by Load Bitmap and load the **bitmap_glyph** field of the DOP Common block with the address of the bitmap in processor memory.

When you specify a bitmap address of 0 and put the actual address in the **bitmap_glyph** field, you save system resources. The bitmap is not loaded until it is accessed, and the application, not the system, is responsible for saving the bitmap (when it is swapped out, it is unknown by the system).

Systemwide Bitmaps

Usually, a bitmap is defined as temporary. That is, the bitmap ID associated with it is not valid once you deassign the channel on which it was loaded. To define a bitmap to last for the life of the system, issue this QIO using the IO$M_QD_SYSTEM_WIDE function modifier.

Notify Deferred Queue Full

Notifies the application when a deferred queue is full.

Format

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

Unique Parameters

P1 — IO$C_QD_DEFERRED_HOLD

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — waiting period

A longword that contains the number of 80-millisecond intervals to wait after the queue is full before notification. (One second is equivalent to seventeen 80-millisecond intervals.)

This parameter must be specified.

P3 — viewport_id

The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P4, P5, P6 — must be 0

Description

The Notify Deferred Queue Full QIO notifies an application when the deferred queue is full. The number of drawing operations that can be deferred for any one viewport is limited so that the system resources cannot be consumed by that viewport. When an application is notified of a full queue, it can load the viewport into memory, execute the deferred operations and delete the deferred queue — freeing up memory. If necessary, the application may then put the viewport back in a deferred state.

Notification may be performed using the QIO AST, an event flag, or a wait.

Read Bitmap

Copies data from video memory to a user-specified buffer in VAX memory and performs bitmap-to-bitmap transfers.

Format

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

Unique Parameters

P1 — buffer address

The address of the buffer in VAX memory into which the driver should copy the bitmap. This buffer must be large enough to hold the bitmap specified by **P4**. For a bitmap-to-bitmap transfer, specify a 0 in this parameter.

This parameter must be specified.

P2 — buffer length

A longword that contains the length of the buffer specified by **P1**, in bytes.

P3 — must be 0

P4 — transfer parameter block

The transfer parameter block (TPB) describes the bitmap to be read into VAX memory. The TPB contains:

- The coordinates for the lower left-hand corner of the bitmap
- The height and width of the bitmap
- A predefined constant indicating the type of transfer
- For a bitmap-to-bitmap transfer, the coordinates for the lower left-hand corner of the target bitmap

The following diagram shows the data structure that specifies the transfer parameters.

TPB$W_X_SOURCE	TPB$B_SIZE	TPB$B_TYPE
TPB$W_WIDTH	TPB$W_Y_SOURCE	
TPB$W_X_TARGET	TPB$W_HEIGHT	
TPB$W_X_TARGET_VEC1	TPB$W_Y_TARGET	
TPB$W_L_TARGET_VEC1	TPB$W_Y_TARGET_VEC1	
TPB$W_Y_TARGET_VEC2	TPB$W_X_TARGET_VEC2	
	TPB$W_L_TARGET_VEC2	

The following list describes the contents of each field in the Transfer Parameter block.

Field	Use
TPB$B_TYPE	Type of transfer being performed, either bitmap-to-processor (BTP) or bitmap-to-bitmap (BTB); use the constants defined below to load this field
TPB$B_SIZE	Reserved to DIGITAL
TPB$W_X_SOURCE	X coordinate of lower left-hand corner of source bitmap
TPB$W_Y_SOURCE	Y coordinate of lower left-hand corner of source bitmap
TPB$W_WIDTH	Width of source bitmap
TPB$W_HEIGHT	Height of source bitmap
TPB$W_X_TARGET	X coordinate of lower left-hand corner of target bitmap (only specified for bitmap-to-bitmap transfer)
TPB$W_Y_TARGET	Y coordinate of lower left-hand corner of target bitmap (only specified for bitmap-to-bitmap transfer)
TPB$W_X_TARGET_VEC1	Reserved to DIGITAL
TPB$W_Y_TARGET_VEC1	Reserved to DIGITAL
TPB$W_L_TARGET_VEC1	Reserved to DIGITAL

Field	Use
TPB$W_X_TARGET_VEC2	Reserved to DIGITAL
TPB$W_Y_TARGET_VEC2	Reserved to DIGITAL
TPB$W_L_TARGET_VEC2	Reserved to DIGITAL

Constants — The following constants are defined in conjunction with the TPB.

Constant	Value
TPB$C_BITMAP_XFR	Used to specify a bitmap-to-bitmap transfer
TPB$C_SOURCE_ONLY	Used to specify a BTP or PTB transfer
TPB$C_SOURCE_LENGTH	Structure length for BTP or PTB transfers
TPB$C_BITMAP_XFR_LENGTH	Structure length for a bitmap-to-bitmap transfer
TPB$C_LENGTH	Full length of TPB structure.

This parameter must be specified.

P5 — TPB length

A longword that contains the length of the transfer parameter block specified in **P4**, in bytes.

This parameter must be specified.

P6 — must be 0

Description

The Read Bitmap QIO reads data from the QDSS video memory into a specified buffer in VAX memory. This function transfers all available planes of memory at once. If this operation is to affect more than one viewport, it must be preceded by a Stop Request Queue QIO.

You can also use it to perform bitmap-to-bitmap transfers. In a bitmap-to-bitmap transfer, you specify a source and target location in the TPB and omit the processor buffer.

Release Hold

Releases all viewports from the hold state.

Format

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

Unique Parameters

P1 — IO$C_QD_NOHOLD

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2, P3, P4, P5, P6 — must be 0

Description

The Release Hold QIO function releases viewports from the hold state.

A viewport is put into the hold state when the Hold Viewport Activity QIO is issued. All viewports, except the systemwide viewport, are affected by a Hold Viewport Activity QIO.

A count of the number of hold requests is maintained. When the number of release hold requests equals the number of hold requests, the driver releases the screen.

Resume Viewport Activity

Resumes activity in a previously suspended viewport.

Format

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

Unique Parameters

P1 — IO$C_QD_RESUME_VP

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — viewport_id

The viewport ID of the targeted viewport. This ID is returned by the Get Viewport ID during the creation of the viewport.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Resume Viewport Activity QIO function resumes activity on a viewport that had previously been suspended using the Suspend Viewport Activity QIO function.

Set Color Characteristics

Identifies a system as either color or intensity based.

Format

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

Unique Parameters

P1 — IO$C_QD_COLOR_CHAR

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — system flag

A longword that indicates whether the system is color or intensity based. A value of 0 indicates a color (RGB) system. A value of 1 indicates an intensity (monochrome) system.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Set Color Characteristics QIO identifies a system as either color or intensity based. Your application must use this call to inform the driver about which type of color system it is using. Once this is specified, the driver only accepts Set Color Map Entries QIO requests that match this setting. It rejects all other Set Color Map Entries QIOs.

Set Color Map Entries

Defines (or alters) the color map.

Format

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

Unique Parameters

P1 — IO$C_QD_SET_COLOR

The function code that identifies the action the QIO performs. This parameter must be specified.

To modify the action of the QIO, you can OR the IO$C_QD_SET_COLOR function code with one of the following optional function modifiers:

Function Modifier	Action
IO$M_QD_INTENSITY	Specify this modifier to set a map entry on an intensity system.
IO$M_RESERVED_COLORS	If the IO$M_RESERVED_COLORS modifier is specified, the starting color map entry (P4) is interpreted as the IO$C_QD_TWO_COLOR_CURSOR (set two-color cursor) parameter. In this case, the buffer must contain two map entries worth of data: two words for intensity systems, six words for color systems.

P2 — address of the color buffer

The color buffer differs depending on whether you are using a color system or an intensity system.

On a color system, the buffer must have an "RGB triple" for each map entry you wish to set. An "RGB triple" is made up of three word-long values, one for red, one for green, and one for blue. So, if you want to set five color map entries, the color buffer must be 15 words long.

On an intensity system, the buffer must have a single word-long value for each map entry you want to set. So, if you want to set five color map entries, the color buffer must be five words long.

Only the eight most significant bits are used for color definition.

This parameter must be specified.

P3 — length of the color buffer, in bytes

A longword that contains the length of the color buffer, in bytes. On color systems, this must be a multiple of 6. On intensity systems this must be a multiple of 2.

This parameter must be specified.

P4 — starting color map entry

A longword that contains the color map index to use for the first RGB or intensity value specified in the buffer.

This parameter must be specified.

P5 — color map selection

Must be 0.

P6 — must be 0

Description

The Set Color Map Entries allows an application to define or alter the color map. Color map information is stored in the QDB. You can access the QDB by issuing the Get System Information QIO. See Appendix B for a full description of the QDB.

Start Request Queue

Starts the processing of any packets on the request queue.

Format

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

Unique Parameters

P1 — IO$C_QD_START

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — viewport_id

The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Start Request Queue QIO function starts (or restarts) the processing of packets on the request queue of the specified viewport.

Typically, your application makes this call after the queue has been stopped with the Stop Request Queue QIO function or after initial viewport creation.

Stop Request Queue

Stops the processing of any packets on the request queue.

Format

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

Unique Parameters

P1 — IO$C_QD_STOP

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — viewport_id

The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Stop Request Queue QIO stops all processing on the request queue of the specified viewport in order to give the calling process complete control over the viewport bitmap. Stopping a viewport's request queue insures that no other processes can modify the bitmap of the stopped viewport.

A viewport management task typically uses this function when changing the position of any viewport in the system (using the Set Viewport Region QIO function). An application can also use it to freeze the screen for printing displayed data.

Once the Stop Request Queue QIO is invoked, no further commands can be executed from the request queue unless the request queue is explicitly restarted with the Start Request Queue QIO.

A Stop Request Queue QIO differs from a Suspend Viewport Activity QIO in that it waits for currently processing DOPs to complete before returning control.

Suspend Occluded Viewport Activity

Suspends all operations to a specified (occluded) viewport when certain operations are requested.

Format

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

Unique Parameters

P1 — IO$C_QD_OCCLUDED_SUSPEND
The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — viewport_id
The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Suspend Occluded Viewport Activity QIO suspends operations to an occluded viewport when a DOP that the driver cannot process is executed.

Operations that the driver cannot perform on occluded viewports are typically Scroll, Move Area, and Move/Rotate operations in which the source is *not* a bitmap ID.

The application must handle the operation and then issue a Resume Viewport Activity QIO to continue request queue processing. You may handle the operation in the associated AST. The address of the DOP is returned in the second longword of the IOSB block. For example, if a Scroll is attempted on an occluded viewport, the driver detects the condition and fires the AST associated with Suspend Occluded Viewport Activity QIO. The AST uses the DOP address returned in the IOSB to determine what action to take and issues a Resume Viewport Activity QIO for the viewport when it completes.

Suspend Viewport Activity

Suspends activity in a specified viewport.

Format

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

Unique Parameters

P1 — IO$C_QD_SUSPEND_VP

The function code that identifies the action the QIO performs.

This parameter must be specified.

P2 — viewport_id

The viewport ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

This parameter must be specified.

P3, P4, P5, P6 — must be 0

Description

The Suspend Viewport Activity QIO function suspends all activity on a specified viewport.

Typically, this call is used to synchronize drawing operations. When any necessary operations are completed, you can resume activity on the viewport by making a call to the Resume Viewport Activity QIO function.

A Suspend Viewport Activity QIO differs from a Stop Request Queue QIO in that it does *not* wait for currently processing DOPs to complete before returning control.

Write Bitmap

Writes data from a user-specified buffer in VAX memory to a bitmap in video memory and also performs bitmap-to-bitmap transfers.

Format

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

Unique Parameters

P1 — buffer address

The address of the buffer in VAX memory from which the bitmap is written. This buffer must be large enough to hold the bitmap specified in the transfer parameter block (P4). In a bitmap-to-bitmap transfer, specify a zero in this parameter.

This parameter must be specified.

P2 — buffer length

A longword that contains the length of the buffer specified in P1, in bytes.

This parameter must be specified.

P3 — must be 0

P4 — transfer parameter block

The transfer parameter block (TPB) that describes the bitmap which is to be written into video memory. The TPB contains:

- The coordinates for the lower left-hand corner of the bitmap
- The height and width of the bitmap
- A predefined constant indicating the type of transfer
- For bitmap-to-bitmap transfer, the coordinates for the lower left-hand corner of the target bitmap

The following diagram shows the data structure that specifies transfer parameters.

TPB$W_X_SOURCE	TPB$B_SIZE	TPB$B_TYPE
TPB$W_WIDTH	TPB$W_Y_SOURCE	
TPB$W_X_TARGET	TPB$W_HEIGHT	
TPB$W_X_TARGET_VEC1	TPB$W_Y_TARGET	
TPB$W_L_TARGET_VEC1	TPB$W_Y_TARGET_VEC1	
TPB$W_Y_TARGET_VEC2	TPB$W_X_TARGET_VEC2	
	TPB$W_L_TARGET_VEC2	

The following list describes the contents of each field in the Transfer Parameter block.

Field	Use
TPB$B_TYPE	Type of transfer being performed, either processor-to-bitmap (PTB) or bitmap-to-bitmap (BTB) Use the constants defined below to load this field
TPB$B_SIZE	Reserved to DIGITAL
TPB$W_X_SOURCE	X coordinate of lower left-hand corner of source bitmap
TPB$W_Y_SOURCE	Y coordinate of lower left-hand corner of source bitmap
TPB$W_WIDTH	Width of source bitmap
TPB$W_HEIGHT	Height of source bitmap
TPB$W_X_TARGET	X coordinate of lower left-hand corner of target bitmap (only specified for bitmap-to-bitmap transfer)
TPB$W_Y_TARGET	Y coordinate of lower left-hand corner of target bitmap (only specified for bitmap-to-bitmap transfer)
TPB$W_X_TARGET_VEC1	Reserved to DIGITAL
TPB$W_Y_TARGET_VEC1	Reserved to DIGITAL
TPB$W_L_TARGET_VEC1	Reserved to DIGITAL
TPB$W_X_TARGET_VEC2	Reserved to DIGITAL
TPB$W_Y_TARGET_VEC2	Reserved to DIGITAL
TPB$W_L_TARGET_VEC2	Reserved to DIGITAL

Constants — The following constants are defined in conjunction with the TPB.

Constant	Value
TPB$C_BITMAP_XFR	Used to specify a bitmap-to-bitmap transfer.
TPB$C_SOURCE_ONLY	Used to specify a BTP or PTB transfer.
TPB$C_SOURCE_LENGTH	Structure length for BTP or PTB transfers.
TPB$C_BITMAP_XFR_LENGTH	Structure length for a bitmap-to-bitmap transfer.
TPB$C_LENGTH	Full length of TPB structure.

P5 — TPB length

A longword that contains the length of the transfer parameter block specified in P4, in bytes.

P6 — must be 0

Description

The Write Bitmap QIO writes data from a specified buffer in VAX memory to QDSS video memory. This function transfers all available planes of memory at once. If this operation is to affect more than one viewport, it must be preceded by a Stop Request Queue QIO.

Your application may also use it to perform bitmap-to-bitmap transfers. In a bitmap-to-bitmap transfer, you specify a source and target location in the TPB and omit the processor buffer.

Chapter 5

Using Drawing Operation Primitives

This chapter describes:

- What drawing operation primitives (DOPs) are
- How to perform drawing operations to the screen using DOPs
- How to perform window management operations using DOPs
- How to use features of the UISDC interface when implementing DOPs

5.1 What Are DOPs?

Drawing operation primitives (referred to as DOPs) are data structures created by your application that contain information the QDSS hardware uses to perform drawing operations to the screen. (Some DOPs are also used to perform window management tasks; for example, to suspend and resume request queue activity on a specific viewport.)

DOPs provide a fast and simple way of performing basic drawing operations. The following is a list of the drawing operations you can perform using DOPs:

- Draw a simple line, a complex line, a series of lines, or a polygon
- Draw a point, or a series of points
- Draw a filled polygon using a bitmap pattern
- Fill points using an associated bitmap pattern
- Move a rectangular area within a viewport
- Move, rotate, and scale a rectangular area within a viewport
- Scroll a rectangular area
- Draw fixed-width text to the screen
- Draw variable-width text to the screen

In addition to drawing operations, you can also perform the following window management operations using DOPs:

- Stop removing entries from a window's request queue
- Start removing entries from a *stopped* request queue
- Suspend drawing operations to a window
- Resume drawing operations in a window

5.1.1 How to Use DOPs

To perform a drawing operation, your application must:

1. Allocate storage for the DOP
2. Define the structure of the DOP
3. Initialize any relevant fields of the DOP structure
4. Execute the DOP

How you structure DOPs differs depending on which operation you wish to perform; initialization varies accordingly.

How you allocate and execute DOPs also differs depending on whether your application uses the UIS windowing environment or provides its own windowing services. If an application uses the UIS environment it can use the UISDC routines described in Section 5.3 to allocate and execute DOPs. If an application does not use the UIS environment, it must allocate and execute DOPs itself. How to do this is described in Section 5.4.

Sections 5.2 through 5.5 provide general information about DOPs that must be understood before you attempt to implement any individual operation. Section 5.6 describes how to structure and initialize DOPs for each type of operation. A FORTRAN example accompanies the explanation of each operation in the section.

5.2 Overview of DOP Structure

This section is a general description of DOP structure that will be useful in understanding how to allocate and execute DOPs. The complete description of how to structure DOPs for specific operations is in Section 5.6.

Each DOP structure is made up of three substructures (blocks):

Common block — A fixed-size block which must begin *all* DOP structures. This block contains information that all DOPs require, for example, the *item_type* field that identifies which operation the DOP performs and the *opcount* field that tells how many times the operation should be repeated.

Unique block — A fixed size block that *all* DOPs require following the Common block. This block contains information that is more specific to a single operation, or group of operations, and its fields and their contents vary accordingly. Some operations do not utilize the fields in this block, others only utilize some of the fields (see the operation-by-operation structure description in Section 5.6 for details) but, regardless of utilization, the DOP structure must be padded with the entire Unique block.

Variable block — A variable-length block that contains operation-specific variables (that is, coordinates, line lengths). The size of this block varies depending on the number of times an operation is repeated. That is, if you specify a draw-line operation with an *opcount* of 1, the Variable block contains the coordinates needed to draw one line. If you specify an *opcount* of 3, then the Variable block must hold the coordinates needed to draw 3 lines (and is three times as long).

Section 5.6 contains an illustration of the Common block along with a full explanation of each field in the block. In addition, the section contains illustrations and explanations of the specific Unique and Variable blocks needed to define DOPs for each type of operation.

5.3 Implementing DOPs Within the UIS Environment

If your application runs in the UIS environment, you may use UISDC routines to allocate and execute DOPs.

5.3.1 Allocating Storage for DOPs in the UIS Environment

To allocate storage for a DOP, use the UISDC$ALLOCATE_DOP routine. This routine allocates memory for the DOP and also initializes some fields of the Common block to default values. UISDC$ALLOCATE_DOP has the following format:

```
dop_address = UISDC$ALLOCATE_DOP (wd_id, size, atb)
```

dop_address	Returned address of the DOP. This address is used in subsequent routines to execute the DOP.
wd_id	Window identifier that you specify. This argument associates the DOP with a window by loading the window-related fields of the Common block (the window dimensions and the clipping rectangle). The window identifier is returned to an application at viewport creation time.
size	Size, in bytes. This argument is read/write - on input, you specify the space for the Variable block; on output, the system returns the actual size allocated for the Variable block. The size allocated may be smaller than the size you request. Always use the *returned* size in subsequent operations.
atb	Address of an attribute block which is used to initialize the color, writing-mode, and writing-mask fields of the Common block.

A full routine description of the UISDC$ALLOCATE_DOP routine appears in Section 5.7.

Example 5–1 is a FORTRAN program segment that creates a window and allocates a DOP to draw 50 points. Note that the input size argument is calculated by multiplying the number of times the DOP repeats the operation (the opcount) by the predefined constant for the length of a single-operation Variable block (DOP_POINTS$C_LENGTH). The SYS$LIBRARY:VWSSYSDEF file defines a constant for the length of a single-operation Variable block for each operation, see Section 5.5.4.1 for information about predefined data structures.

Example 5–1 Allocating a DOP

```
        .
        .
        .
! Create a display and window
VD_ID = UIS$CREATE_DISPLAY (0.0,0.0,     ! lower left corner
2                           50.0,50.0,   ! upper right corner
2                           15.0,15.0)   ! width & height

WD_ID = UIS$CREATE_WINDOW (VD_ID,                        ! display ID
2                          'SYS$WORKSTATION',            ! device name
2                          'DOP Drawing Window')         ! window banner

! Allocate the DOP
SIZE = (50 * DOP_POINTS$C_LENGTH)        ! variable block
DOP = UISDC$ALLOCATE_DOP (WD_ID,         ! window ID
2                         SIZE,          ! size, in bytes
2                         0)             ! default ATB number
```

5.3.1.1 The Allocation Mechanism

The mechanism the system uses for allocating DOPs provides for efficient use of storage in the following ways:

- It allocates a small amount of storage, if possible.
- It reuses the storage occupied by DOPs once they have been executed.
- It waits for storage to be freed when too much is allocated.

DOPs are allocated in two sizes, by default — 128 bytes (small) and 786 bytes (large). You may set the DOP sizes and the number of available DOPs. (See the following section.) When you specify the size of the variable portion of the DOP you wish to allocate, the system calculates whether it can allocate a small DOP, and does so if it can. Otherwise, it allocates a large DOP.

The system also reuses any storage occupied by an already-executed DOP. Once a DOP is executed, it is taken off the *request queue* (see the next section) and put on a *return queue*. A return queue is a data structure that points to a linked list of previously executed DOPs (the DOP Common block provides forward and backward links). Each viewport is associated with two return queues; one for small DOPs and one for large DOPs. The viewport ID is an address that points to a data structure that holds both associated return queues.

When you allocate a DOP using UISDC$ALLOCATE_DOP, the system first checks whether a small or large DOP is required. It then attempts to reuse any previously executed DOPs of the needed size. If none exists, it allocates a DOP from system memory or waits until a DOP is free for reuse.

5.3.1.2 Modifying DOP Size and Number

By default, the available sizes and numbers of DOPs are the following:

- 300 small DOPs; each 128 bytes long
- 150 large DOPs; each 768 bytes long

You may use these values or you may wish to alter them to suit your needs. If change these values, you must do so before any other processing is done. That is, you cannot start using DOPs of one size then change the size and continue.

You may modify the default sizes of the small and large DOPs with the following restrictions:

- Small DOPs must be within the range 128-512, inclusive.
- Large DOPs must be within the range 768-16384, inclusive.

You may also modify the number of available DOPs with the following restriction:

- There must be a minimum of 110 small DOPS and 110 large DOPs.

Note that the default DOP setup allocates 300 pages of P1 address space for DOPs. If your modifications to size and number of DOPs results in the need for more than 300 pages, do the following:

1. Increase the logical value UIS$P1_POOL_SIZE by the increased size *in bytes.*
2. Increase the SYSGEN parameter, CTLPAGES by the increased size *in pages.*

To modify the size and number of DOPs, redefine the following logical values, as appropriate.

UIS$SMALL_DOP_SIZE	Size of a small DOP. Default is 128.
UIS$LARGE_DOP_SIZE	Size of a large DOP. Default is 768.
UIS$NUMBER_OF_SMALL_DOPS	Number of available small DOPs. Default is 300.
UIS$NUMBER_OF_LARGE_DOPS	Number of available large DOPs. Default is 150.

5.3.2 Executing DOPs in the UIS Environment

Once you have defined, allocated and initialized a DOP, you are ready to execute it. You have three options when executing DOPs:

1. Execute the DOP synchronously, using the UISDC$EXECUTE_DOP_SYNCH routine.
2. Execute the DOP asynchronously with completion notification, using the UISDC$EXECUTE_DOP_ASYNCH routine.
3. Execute the DOP asynchronously *without* completion notification, using the UISDC$QUEUE_DOP routine.

The difference between the two types of asynchronous execution is that the UISDC$EXECUTE_DOP_ASYNCH routine takes an I/O status block (IOSB) that your application can use to check for completion notification. UISDC$QUEUE_DOP queues the DOP for execution, but provides no way of telling when the operation is completed (it is more efficient). Complete descriptions of these routines and their arguments appears in Section 5.7.

PERFORMANCE NOTE: The UISDC$QUEUE_DOP routine is much more efficient than the other routines; use it whenever an option exists.
The following FORTRAN program segment defines and initializes a DOP using a subroutine (the recommended method for FORTRAN) and queues a DOP for asynchronous execution without completion notification.

Example 5–2 Queuing a DOP for Execution

```
! Allocate the DOP
        .
        .
        .
! Call subroutine to initialize DOP
CALL SUB_STRUCTURE (%VAL(DOP),                ! address of DOP
                    %VAL(DOP+DOP$C_LENGTH)) ! address of a variable
                                              ! block

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,              ! window ID
2                      %VAL(DOP))         ! DOP address, by value
```

5.3.2.1 The Execution Mechanism

DOPS are executed using a mechanism called a *request queue*. Each viewport has a request queue associated with it. A request queue is simply a linked list of any DOPs that have been submitted to a viewport for execution. Viewport IDs (including the systemwide viewport ID) are actually addresses that point to the request queue structure associated with the viewport. The request queue structure contains the starting address of the linked list of DOPs. Each DOP contains forward and backward links (in its Common block) to other DOPs in the list. These links are updated by the system when a DOP is submitted or executed.

When you execute a DOP, you must provide the above-mentioned routines with the *window ID* of the viewport in which you wish to execute the DOP (returned in the UIS$CREATE_WINDOW call) and the *address* of the DOP (which is returned by UISDC$ALLOCATE_DOP). These arguments provide the system with the information needed to associate the DOP with the correct request queue.

5.4 Implementing DOPs in a Non-UIS Environment

If your application does not make use of the UIS windowing system; for example, if it provides its own windowing system, it must allocate and execute DOPs itself. The following sections describe how an application allocates storage for DOPs and how it inserts them on the request queue for execution.

5.4.1 Allocating Storage for DOPS in a Non-UIS Environment

Before allocating new storage for a DOP you should check the associated return queue to determine if any reusable storage is available on the queue. Use the viewport ID to access the return queue —- it is the address of the DOP Queue structure (in which the return queue resides). See Appendix B for a full description of the data structure. Remember, there are actually two return queues associated with each viewport one for the large DOPs and one for the small DOPs. The size of a DOP depends on the amount of information needed to fully describe the requested drawing operation(s). Use the MACRO REMQUE instruction to remove a DOP from the return queue.

If no reusable storage exists, allocate storage using the LIB$GET_VM routine.

The following example creates a viewport then allocates a DOP by first checking the return queue. Note that the MACRO module must be compiled separately and linked with a FORTRAN program.

Example 5–3 Allocating a DOP

```
PROGRAM DELETE_VIEWPORT
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
INCLUDE '($IODEF)'

INTEGER*2 CHAN_VP1

! Declare URD
INTEGER*2 URD1_VP1(6)

! Load URD1_VP1 buffer
URD1_VP1(1) = 0          ! lower left corner
URD1_VP1(2) = 0
URD1_VP1(3) = 99         ! upper right corner
URD1_VP1(4) = 99
URD1_VP1(5) = 10         ! absolute coordinate base
URD1_VP1(6) = 10

! Define and start VP1
CALL VIEWPORT (URD1_VP1, CHAN_VP1, VP1_ID)

! Get a Draw Lines DOP for VP1
SIZE = (4 * DOP_LINE$C_LENGTH)          ! calculate size
CALL GET_DOP (VP1_ID, SIZE, DOP1)
            .
            .
            .
! **********************
! * Get DOP Subroutine *
! **********************
SUBROUTINE GET_DOP (VIEWPORT_ID, SIZE, DOP)
IMPLICIT INTEGER*4(A-Z)

! Declare external MACRO routine
EXTERNAL DOP$REMQUE

DOP = DOP$REMQUE (VIEWPORT_ID,
2                 SIZE)
```

(Continued on next page)

Example 5–3 (Cont.) Allocating a DOP

```
        ! If none on return queue, calculate size and allocate one.
        IF (DOP .EQ. 0) THEN
           CALL TEST_SIZE (%VAL(VIEWPORT_ID),    ! viewport ID > return Q
        2                  SIZE)
           ! Allocate appropriate size DOP
           CALL LIB$GET_VM (SIZE,
        2                   DOP)
        END IF

        RETURN
        END

        ! ************************
        ! * TEST_SIZE SUBROUTINE *
        ! ************************

        SUBROUTINE TEST_SIZE (REQ,SIZE)
        IMPLICIT INTEGER*4(A-Z)
        INCLUDE 'VWSSYSDEF'

        ! Associate the predefined structure w/ REQ
        RECORD /REQ_STRUCTURE/ REQ

        IF (SIZE .GT. REQ.REQ$W_SMALL_DOP_SIZE) THEN
           SIZE = REQ.REQ$W_LARGE_DOP_SIZE
        ELSE
           SIZE = REQ.REQ$W_SMALL_DOP_SIZE
        END IF

        RETURN
        END
;
;       Separate MACRO Module
;

        .title rem_que - does a remque

        $DOPDEF
        $REQDEF
```

(Continued on next page)

Example 5-3 (Cont.) Allocating a DOP

```
;++
; dop$remque - remove a DOP from the return queues and return it
;
; description:
;       This routine will return a DOP or zero if none is
; available.  The size is used to determine whether to use a large
; or small DOP.  Note: that it is possible that the size is larger
; than the DOP returned, if this is the case then the application
; must break the request down into smaller chunks and use several
; large DOPs.
;
; Calling sequence:
;       DOP = DOP$REMQUE(VIEWPORT_ID,SIZE)
;
; Outputs:
;       DOP = ZERO if no DOPS available on the queue
;--
        .entry  dop$remque,0
        movl    @4(ap),r1                       ; get req address
        cmpw    @8(ap),req$w_small_dop_size(r1) ; check for small or large DOP
        bgtr    10$                             ;
        remque  @req$l_return_flink(r1),r0      ; try to get a DOP
        bvs     90$                             ; nope then clear r0 and return
        ret
;
; We need a large DOP
;
10$:    remque  @req$L_return_large_flink(R1),r0; get a large DOP
        bvs     90$                             ; none of these then return
        ret                                     ; all set then return
;
; Can't get the DOP we want
;
90$:    clrl    r0                              ; signal error (return zero)
        ret
        .end
```

5.4.2 Executing a DOP in a Non-UIS Environment

To execute a DOP in a non-UIS environment, an application must insert the DOP on the request queue. To insert a DOP on the request queue, use the MACRO INSQUE instruction. Use the viewport ID to access the request queue — it is the address of the DOP Queue structure (in which the return queue resides). See Appendix B for a full description of the data structure.

The following example creates a viewport, allocates a DOP, loads the DOP with the necessary information, and inserts the DOP on the request queue. Note that the MACRO module must be compiled separately and linked with a FORTRAN program.

Example 5-4 Inserting a DOP on the Request Queue

```
      .
      .
      .
          ! Declare external MACRO routine
          EXTERNAL DOP$INSQUE
      .
      .
      .
          ! Define and start VP1
          CALL VIEWPORT (URD1_VP1, CHAN_VP1, VP1_ID)

          ! Get a Draw Lines DOP for VP1
          SIZE = (4 * DOP_LINE$C_LENGTH)          ! calculate size
          CALL GET_DOP (VP1_ID, SIZE, DOP1)

          ! Call the draw lines subroutine for a border
          CALL D_LINES (%VAL(DOP1),               ! DOP address, by value
          2             %VAL(DOP1+DOP$C_LENGTH),  ! var. block address
          2             SIZE)                     ! DOP size

          ! Queue the DOP by calling a MACRO subroutine
          CALL DOP$INSQUE (%VAL(DOP1),    ! DOP address, by value
          2                VP1_ID)        ! viewport ID
      .
      .
      .
;
;         Separate MACRO Module
;
          .title ins_que - does an insque tail

          $DOPDEF
          $REQDEF
```

(Continued on next page)

Example 5–4 (Cont.) Inserting a DOP on the Request Queue

```
;++
; dop$insque - inserts a DOP at the tail of the return queue
;
; calling sequence:
;
;       CALL DOP$INSQUE(DOP,VIEWPORT_ID)
;
; OUTPUTS:
;       NONE
;--
.entry  dop$insque,0
        movl    4(ap),r0
        movl    @8(ap),r1
        insque  (R0),@req$L_request_blink(R1)
        movl    #1,r0                   ; indicate success
        ret

.end
```

5.5 Structuring and Initializing DOPs

As mentioned briefly above, each DOP structure is made up of three blocks — the Common block, the Unique block, and the Variable block. The fields of the Common block are the same across operations, while those of the Unique and Variable blocks vary depending on the operation being performed.

The later part of this section (Section 5.6) is designed to provide easy reference for programming DOPs. It contains the following:

- An illustration and description of the Common block
- Separate illustrations and descriptions of the Unique block and Variable block for each operation
- FORTRAN examples accompanying the description of each operation

For quick reference, the running head of each page reflects the operation being described on that page.

Before proceeding to Section 5.6, read the following sections for general information about structuring DOPs and using the reference section.

5.5.1 The Common Block

The Common block is a fixed-size block that must begin *all* DOP structures. It contains a number of fields (described completely in Section 5.6), but two in particular affect the subsequent structure of the DOP — the *item_type* field and the operation count *opcount* field.

The *item_type* field determines which operation the DOP performs, and so affects which Unique and Variable blocks must be specified. You must explicitly specify the item type using a predefined constant (defined in SYS$LIBRARY:VWSSYSDEF). The table that follows lists the symbolic constants used to specify all possible drawing operations along with a brief description of each.

Constant	Operation
DOP$C_DRAW_LINES	Draw line(s) or polygon.
DOP$C_DRAW_COMPLEX_LINE	Draw a complex patterned line with a sloped length and width. Also draws a filled polygon.
DOP$C_DRAW_POINTS	Draw point(s).
DOP$C_FILL_POLYGON	Draw a filled polygon with a specified pattern.
DOP$C_FILL_POINT	Fill a point with a specified pattern.
DOP$C_FILL_LINES	Fill a line with a specified pattern.
DOP$C_DRAW_FIXED_TEXT	Draw text with a specified fixed-space font.
DOP$C_DRAW_VAR_TEXT	Draw text with a specified variable-spaced font.
DOP$C_MOVE_ROTATE_AREA	Move an area with specified rotation and scaling.
DOP$C_MOVE_AREA	Move an area within a viewport.
DOP$C_SCROLL_AREA	Move area in a viewport. Fill the vacated area with background color
DOP$C_STOP	Stop removing entries from this request queue. This used when manipulating the screen to handle occlusion.
DOP$C_START	Start removing entries from the specified request queue. This request can only be made from the system viewport.
DOP$C_DELETE_BITMAP	Delete an offscreen bitmap.
DOP$C_SUSPEND	Do not remove any more DOPs from this request queue until the queue is resumed.
DOP$C_RESUME	Resume processing DOPs on the specified queue. This request can only be made from the system viewport.

The *opcount* field indicates how many times the operation should be repeated. That is, if the specified *item_type* is dop$c_draw_lines and the *opcount* is 4, the DOP will draw four lines. This affects the structure of the DOP because the size of the

Variable block varies with the number of operations (more coordinates are necessary to perform the additional operations).

5.5.2 The Unique Block

The Unique block is a fixed-size block. This block contains information that is, specific to a single operation or group of operations, and its fields and their contents vary accordingly. Every DOP structure must include the Unique block whether or not it utilizes the fields in the block. That is, if a particular operation only uses three words of the Unique block, the Unique block of the DOP structure must be padded to its full length.

Section 5.6 illustrates and explains the fields in the Unique block that are relevant for each operation.

5.5.3 The Variable Block

The Variable block is a variable-size block. It contains operation-specific variables (that is coordinates, line lengths). The size of the block varies depending on the number of operations the DOP is to perform. To clarify, if you specify a draw-line operation with an *opcount* of 1, the Variable block contains the coordinates needed to draw one line. If you specify an *opcount* of 3, then the Variable block must hold the coordinates needed to draw 3 lines (and be three times as long).

Section 5.6 illustrates and explains the fields in the Variable block that are needed for a single occurrence of each operation. It also lists the predefined constant for the length of a one-operation Variable block. This constant is useful for calculating the input size argument of the UISDC$ALLOCATE_DOP routine. Simply multiply the constant by the number of times the DOP is to repeat the operation (the *opcount*).

5.5.4 Programming Considerations

When programming an application that performs drawing operations, you have several options for how to implement the DOP structure. Depending on the language in which you are writing and the exact nature of the operation, you can:

- Use the predefined structure provided in the SYSLIBRARY:VWSSYSDEF file
- Define the full DOP structure using your language's structured-type statements

Using the predefined DOP structure prevents having to explicitly construct the DOP in your application. However, in some cases, it is not always possible to use the predefined structure.

5.5.4.1 The Predefined DOP Structure

The system provides a DOP definition file located in SYSLIBRARY:VWSSYSDEF.*lan* - where *lan* is the file extension for the language you are using. (You choose which language definition files are created when you install the system.)

VWSSYSDEF defines DOP-related constants, including offset values that define each field in the DOP structure. You can use these predefined offsets in your application to initialize the DOP fields. Remember, the system allocates the DOP storage; you must associate a structure with the storage in order to initialize the proper fields. The offsets provide a way of accessing fields within the structure. Section 5.6 labels each field in the DOP illustrations with its predefined offset.

For example, VWSSYSDEF defines an offset DOP$W_ITEM_TYPE which identifies the location of the *item_type* field in the DOP. Once you associate the returned storage with a structure, you can use the offset to reference into the structure.

You must "include" or "insert" the VWSSYSDEF file in each module in which you reference it in order to use any predefined constants and offsets. You should become familiar with the way in which the VWSSYSDEF file defines the DOP structure for the programming language you are using.

Initializing fields using the offsets is straightforward when applied to the fixed portion of the DOP (the Common and Unique blocks), but is somewhat more complex when applied to the Variable block. The VWSSYSDEF file only defines offsets for one occurrence of an operation in the Variable block and they are offset from the end of the fixed portion. To clarify, to draw a single line, the Variable block for a Draw Lines operation requires four fields, which have predefined offsets; however, to draw two lines, it requires eight fields — *but only four offsets are predefined.* To write to the second set of fields, you can use the same offsets, *but you must change the point from which they are offset.*

In languages that support arrays of structures, such as PASCAL, you can define an array of Variable block structures, and increment the original offset by the length of a "one-operation" Variable block. The VWSSYSDEF file provides constants for both the offset and Variable block length. The original offset (the end of the fixed portion) is DOP$C_LENGTH and the constants for the "one-operation" Variable block lengths are listed with the respective operation in Section 5.6.

In a language that does *not* support arrays of structures, such as FORTRAN, you have two options:

- Completely define the structure of the Variable block using a STRUCTURE statement.
- Define a structure the size of a one-operation Variable block. Then, to initialize the DOP for series of operations, use a loop that increments the offset.

For a one-operation case, FORTRAN can simply use the predefined offsets.

The examples that accompany the operation descriptions are written in FORTRAN; the following section describes the conventions that they follow.

5.5.4.2 Using the Examples

The examples provided in the DOP reference section are in FORTRAN. They are subroutines that construct the DOP structure and initialize any necessary fields. To eliminate redundancy, the calling program does not appear in each example. However, a typical calling program appears below for the sake of illustration. The sample calling program does the following:

- Creates a display and window.
- Allocates storage for a DOP that draws a box (4 lines — opcount equals 4).
- Passes the returned DOP address to a subroutine that defines and initializes the DOP structure. (This must be done in FORTRAN to avoid reallocating storage during the structure declaration — other high-level languages can avoid the subroutine call by using a pointer to associate the structure with the allocated storage.) It also passes the address of the Variable block (which is calculated by adding the length of the fixed portion of the DOP to the DOP address).
- Queues the DOP for execution.
- Hibernates the process (so the result can be viewed).

NOTE: The examples assume that the application is using the UIS/UISDC interface.

Example 5-5 Calling Program for Example Subroutines

```
PROGRAM DRAW_LINES
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'UISENTRY'
INCLUDE 'VWSSYSDEF'

! Create a display and window
VD_ID = UIS$CREATE_DISPLAY (0.0,0.0,     ! lower left corner
2                           50.0,50.0,   ! upper right corner
2                           15.0,15.0)   ! width & height

WD_ID = UIS$CREATE_WINDOW (VD_ID,                  ! display ID
2                          'SYS$WORKSTATION',      ! device name
2                          'DOP Drawing Window')   ! window banner

! Allocate the DOP
SIZE = (4 * DOP_LINES$C_LENGTH)
DOP = UISDC$ALLOCATE_DOP (WD_ID, ! window ID
2                         SIZE,  ! variable portion size, in bytes
2                         0)     ! default ATB number

! Call the subroutine
CALL SUB_STRUCTURE (%VAL(DOP),              ! DOP address
                    %VAL(DOP+DOP$C_LENGTH))! Var. block address

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,             ! window ID
2                     %VAL(DOP))         ! DOP address, by value

CALL SYS$HIBER()
END
```

NOTE: In FORTRAN, you *must* pass the DOP address that UISDC_$ALLOCATE returns *by value* to a subroutine. The subroutine associates the address with a record structure using the RECORD statement. If this were not done in a subroutine, the RECORD statement would redundantly allocate storage for the DOP.

5.6 The DOP Reference

This section is designed to provide an easy reference for structuring DOPs.

It contains the following:

- An illustration and detailed description of the the Common block structure
- An illustration and detailed description of the Unique and Variable block structure *for each operation*
- A description of how to perform each operation

- A program example of how to perform each operation

The Common block is described first, then each operation is described separately. The operations are listed in alphabetical order. The running head reflects which DOP is described on a page.

Each block is accompanied by the record name associated with the block in the SYS$LIBRARY:VWSSYSDEF definition file, and each field lists the predefined field name. These names can be used to reference into the DOPs.

NOTE: The descriptions assume that an application is using the UIS/UISDC interface to allocate and execute DOPs and to load bitmaps. You may perform these functions without using the UIS/UISDC interface.

Common Block

This is a description of the Common block of the DOP structure. The structures of *all* DOPs must begin with the Common block.

Common block (dop_structure)

DOP$L_FLINK		
DOP$L_BLINK		
DOP$B_SUB_TYPE	DOP$B_TYPE	DOP$W_SIZE
DOP$L_DEC_RESERVED		
DOP$L_DEC_RESERVED2		
DOP$L_USER_RESERVED		
DOP$W_OPCOUNT		DOP$W_FLAGS
DOP$W_MODE		DOP$W_ITEM_TYPE
DOP$L_MASK		
DOP$L_SOURCE_INDEX		
DOP$L_FCOLOR		
DOP$L_BCOLOR		
DOP$L_BITMAP_ID		
DOP$L_BITMAP_GLYPHS		
DOP$W_VP_MAX_Y		DOP$W_VP_MAX_X
DOP$W_DELTA_Y		DOP$W_DELTA_X
DOP$W_VP_MIN_Y		DOP$W_VP_MIN_X

Field	Description
DOP$L_FLINK	This field contains the DOP queue forward link, which is the address of the preceding DOP in the request queue.
DOP$L_BLINK	This field contains the DOP queue backward link, which is the address of the subsequent DOP in the request queue.
DOP$W_SIZE	Size of the DOP in bytes.
DOP$B_TYPE	Structure type of the DOP.
DOP$B_SUB_TYPE	This field is reserved for use by DIGITAL.
DOP$L_DEC_RESERVED	This field is reserved for use by DIGITAL.
DOP$L_DEC_RESERVED2	This field is reserved for use by DIGITAL.
DOP$L_USER_RESERVED	This field is reserved for use by the user.
DOP$W_FLAGS	The following fields are defined within *DOP$W_FLAGS*: DOP$V_DELETE_BITMAP — This field is 1 bit long, and starts at bit 0. When this field contains a value of 1, the offscreen source bitmap (identified in the *bitmap_ID* field) is deleted when the operation completes. DOP$V_SYSTEM_DOP — This field is 1 bit long and starts at bit 1. When this field contains a value of 1, the DOP is returned to the *system* return queue when the drawing operation is completed. A window manager specifies this when it needs to draw in the systemwide viewport or when it allocates a system DOP to draw on a user's viewport. DOP$V_NO_RETURN — This field is 1 bit long, and starts at bit 2. When this field contains a value of 1, the DOP is not returned to the return queue when the drawing operation is completed and is not deleted. This is useful in cases where an application wishes to preserve information in the DOP, possibly in the user-reserved field.
DOP$W_OPCOUNT	Number of drawing operations requested by this packet. For example, if the drawing operation you request is "draw points", this field indicates how many points should be drawn (using coordinates in the variable portion of the DOP).

Field	Description
DOP$W_ITEM_TYPE	Type of drawing operation requested by DOP. The following is a list of the symbolic constants used to specify all possible drawing operations: DOP$C_DRAW_LINES DOP$C_DRAW_COMPLEX_LINE DOP$C_DRAW_POINTS DOP$C_FILL_POLYGON DOP$C_FILL_POINT DOP$C_FILL_LINES DOP$C_DRAW_FIXED_TEXT DOP$C_DRAW_VAR_TEXT DOP$C_MOVE_ROTATE_AREA DOP$C_MOVE_AREA DOP$C_SCROLL_AREA DOP$C_STOP DOP$C_START DOP$C_DELETE_BITMAP DOP$C_SUSPEND DOP$C_RESUME
DOP$W_MODE	Writing mode code as expected by QDSS. There are 16 different writing modes that you can specify using constants listed in Appendix C.
DOP$L_MASK	This field must be -1.
DOP$L_SOURCE_INDEX	Source index. Used with the writing mode to determine the result of overlaid colors. (See the Description section.)
DOP$L_FCOLOR	Foreground color index. This is an index into the color map that is used for the foreground (or writing) color.
DOP$L_BCOLOR	Background color index. This is an index into the color map that is used for the background color.
DOP$L_BITMAP_ID	ID bitmap. Used to identify bitmaps in text images, pattern fill, and delete bitmap operations. This value is 0 if the operation is not related to bitmaps.
DOP$L_BITMAP_GLYPHS	Bitmap backing store address. If a stored bitmap is paged out of the bitmap storage area, the address at which it is stored in VAX memory is loaded into this field.
DOP$W_VP_MAX_X	Viewport relative device coordinate maximum X. Used to determine the upper right corner of the viewport.
DOP$W_VP_MAX_Y	Viewport relative device coordinate maximum Y. Used to determine the upper right corner of the viewport.

Field	Description
DOP$W_DELTA_X	Delta from the lower left corner of the viewport to the lower left corner of the clipping rectangle (the actual writing area).
DOP$W_DELTA_Y	Delta from the lower left corner of the viewport to the lower left corner of the clipping rectangle (the actual writing area).
DOP$W_VP_MIN_X	Viewport relative device coordinate minimum X. Used to determine the lower left corner of the viewport.
DOP$W_VP_MIN_Y	Viewport relative device coordinate minimum Y. Used to determine the lower left corner of the viewport.

Description

The Common block is a fixed-size block that must begin *all* DOP structures. A number of the fields in this block must be loaded in every DOP, while other fields in the block are important only to specific types of operations (for example, color operations and bitmap operations).

Required fields

The number of fields you must explicitly load will differ, depending on whether your application is running in the UIS environment.

If your application is not running in the UIS environment, it is responsible for loading all relevant fields of the Common block. That is, any field needed for the particular DOP except for the first six fields — is loaded by the system during allocation. Section 2.16.2 contains an example that initializes the entire Common block.

If your application runs in the UIS environment, some of the Common block fields that must be initialized are actually loaded by UISDC$ALLOCATE_DOP after it allocates storage for the DOP, while others must be explicitly initialized by your application. The following sections list the fields in these two categories.

Default Initialization — UISDC$ALLOCATE_DOP initializes some fields, using the attribute block the application specifies in the routine call. They are the following:

- Writing mode
- Foreground and background colors
- Viewport minimum and maximum coordinates
- Clipping rectangle delta coordinates

You may affect any of these fields by modifying the ATB before the allocation or by directly overriding the initialization after allocation.

The following fields, initialized by the ALLOCATE routine, are used for the DOP execution mechanism:

- Forward link
- Backward link
- Size
- Type
- Sub_type

While you may access these fields, you should not attempt to modify them.

Explicit Initialization — The fields that your application must explicitly initialize are the *item_type* field and the *opcount* field.

The *item_type* field identifies which operation the DOP performs. You specify one of the symbolic constants listed above to initialize this field.

The *opcount* field tells how many times the operation should be repeated. That is, if you specify a Draw Lines operation with an *opcount* of 1, one line is drawn. If you specify an *opcount* of 3, three lines are drawn. This field is directly related to the Variable block of a DOP. The Variable block contains the coordinates needed to draw a line (in this example). To draw three lines, the Variable block must hold the coordinates needed to draw three lines (and be three times as long).

Color Fields

The following fields in the Common block may be used to manipulate color:

- Foreground color index
- Background color index
- Writing mode
- Source index

The *foreground color index* and *background color index* determine the color to use for writing and background, respectively. Modifying the index changes the color on a per-operation basis. That is, DOPs can write in different colors to the same viewport.

The *writing mode* is used to determine how writing operations use foreground and background colors to display graphic objects; for example, whether objects overlay one another on the screen or negate each other. There are 16 writing modes (see Appendix C).

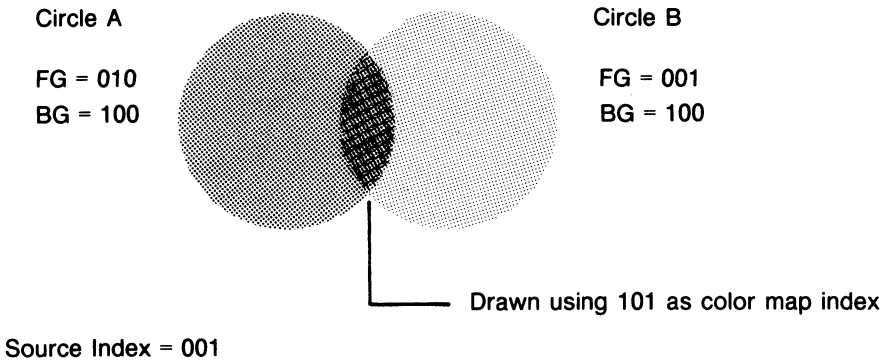
The *source index* is an index number that is used to determine the color interaction of objects being written to the screen with objects already existing on the screen. The source index involves the interaction of:

- The state of the pixels in existing bitmap
- The specified source index value
- The specified writing mode
- The specified foreground and background colors.
- The *use_mask* modifier of the writing mode (if specified)
- The specified source bitmap (if *bitmap_id* field is specified)

The following figure demonstrates how the source index works. (This figure does not specify the source bitmap or the *use_mask* writing mode modifier.)

Figure 5–1 illustrates two intersecting circles. Circle A is written with a foreground index of 010 (binary — as are all numbers in this example) and a background index of 100 (remember, the indexes represent colors in the color map — to simplify this example, only three bits are used to represent pixel settings). Circle B is written with a foreground index of 001 and a background index of 100. The specified source index value is 001 and the specified writing mode is WRIT$DSO (destination ORed with source).

Figure 5–1 How the Source Index Works



When objects are written to the screen, the system performs a logical operation on the present contents of the screen bitmap and the source index. The logical operation is determined by the specified writing mode. Because the DSO mode was specified in Figure 5–1, Circle A's foreground index 010 is ORed with the source index, 001, resulting in the number 011.

The system uses the following procedure to determine the color map index to use for writing:

- If the bit position in the resulting number includes a 1, the system checks the corresponding bit position in the foreground index and sets or clears the bit to agree with the foreground index.
- If the bit position in the resulting number includes a 0, the system checks the corresponding bit position in the background index and sets or clears the bit to agree with the background index.

In Figure 5–1, the number resulting from the logical operation is 101, so the intersecting portion of the circles is written in the color represented by 101 in the color map.

When you specify a bitmap and/or the use_mask writing mode modifier, this process gets more complicated. The following equations describe how to determine the result of any intersection on the screen. (In Figure 5–1, steps 2 and 5 are not meaningful because these options are not specified.)

STEP 1

```
                IF (source_bitmap exists AND
bitmap_index =       source_bitmap = 0) THEN
                   0
                ELSE
                   source_index
```

If you specify a bitmap to be used by the DOP (with the bitmap ID), the driver creates a bitmap index by inserting the source index in any bit position where a 1 occurs in the bitmap. The *source_bitmap* is 1 bit while the resulting *bitmap_index* reflects the number of planes of color used.

STEP 2

```
mask = IF use_mask THEN
          bitmap_index
       ELSE
          -1
```

If you specify the use_mask modifier as part of the writing mode, the bitmap_index is used as the mask.

STEP 3

```
                              IF use_mask THEN
(data)  logical operation        source_index   = fg_bg_selector
                              ELSE
                                 bitmap_index
```

To attain the foreground and background selector perform a logical operation on the data on the screen and either the *source_index* value (if *use_mask* was specified) or the *bitmap_index*. The logical operation is determined by the specified writing mode.

STEP 4

```
( fg AND fg_bg_selector) OR (bg AND (NOT fg_bg_selector)) = t1
```

The value of the foreground and background selector determines which bits are set to foreground and background colors — fg and bg refer to the foreground and background colors specified in the DOP. (t1 is the data output to screen in cases where no mask is specified.)

STEP 5

```
(t1 AND mask) OR (data AND (NOT(mask)) = final_data
```

The mask and t1 are used to determine the data output to the screen.

Bitmap Fields

Two fields in the Common block are used with bit map operations. (Bitmap operations may be text or fill-pattern operations.) They are the *bitmap_ID* field and the *bitmap_glyphs* field.

An operation that uses a bitmap must first load the bitmap from processor memory into offscreen bitmap memory, using the UISDC$LOAD_BITMAP routine. That routine returns a *bitmap_ID* that identifies the loaded bitmap in offscreen memory.

To use that bitmap in a bitmap-related DOP, your application must initialize the *bitmap_ID* field with the returned ID. For example, once you have loaded a bitmap and initialized the *bitmap_ID* field of a Fill Polygon DOP, the DOP will use the bitmap pattern to fill the polygon it creates.

The *bitmap_glyphs* field allows the system to retrieve the bitmap that is not available in offscreen memory. The *bitmap_glyph* is the address (in processor memory) at which the bitmap is stored. If a DOP requires the bitmap, it is swapped back in.

NOTE: The *bitmap_glyph* address must remain valid and not be reused until a Delete Bitmap DOP has been executed and that DOP has completed.

Miscellaneous Fields

The *writing mode* field specified in a DOP affects the way the drawing operations appear on the screen. On a QDSS system, the screen is more than 1 bit deep. Some operations (such as Move Area) may require a writing mode other than the default to operate properly.

By setting bits in the *flags* field, you may do the following:

- Delete the bitmap specified in the *bitmap_ID* field. This is typically done for a "one-shot" bitmap DOP. The DOP is performed, then the bitmap is deleted from offscreen memory.
- Not return a DOP for reuse. Again this is typically done for a "one-shot" DOP. This prevents information contained in the DOP from being overwritten.
- Return the DOP storage to the *systemwide* return queue. This is done when a DOP is allocated from the systemwide viewport and inserted on another viewport's request queue.

Examples

The examples in the operation-specific sections show how to initialize the Common block for each drawing operation.

Initializing the Variable Block

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Delete Bitmap

The Delete Bitmap operation permits you to delete a bitmap specified in the *bitmap_ID* field of the common block.

Unique block

The Unique block is not relevant to this operation.

Variable block

The Variable block is not relevant to this operation.

Description

The Delete Bitmap operation permits you to delete a bitmap that you specify in the *bitmap_ID* field of the Common block.

Since this is a queued operation, make sure that bitmap glyphs being maintained by the application remain valid until Delete Bitmap has been executed.

Relevant Common Block Fields

You specify the bitmap ID of the bitmap you wish to delete in the Common block.

Initializing the Variable Block

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Draw Complex Line

The Draw Complex Line operation permits you to draw a line or patterned line of variable width with a slope in both the length and width direction.

Unique block

The Unique block is not relevant to this operation.

Variable block (dop_move_r_array)

DOP_MOVE_R$W_X_SOURCE	DOP_MOVE_R$W_FILLER
DOP_MOVE_R$W_WIDTH	DOP_MOVE_R$W_Y_SOURCE
DOP_MOVE_R$W_X_TARGET	DOP_MOVE_R$W_HEIGHT
DOP_MOVE_R$W_X_TARGET_VEC1	DOP_MOVE_R$W_Y_TARGET
DOP_MOVE_R$W_L_TARGET_VEC1	DOP_MOVE_R$W_Y_TARGET_VEC1
DOP_MOVE_R$W_Y_TARGET_VEC2	DOP_MOVE_R$W_X_TARGET_VEC2
	DOP_MOVE_R$W_L_TARGET_VEC2

Field	Use
DOP_MOVE_R$W_FILLER	This field is reserved for use by DIGITAL.
DOP_MOVE_R$W_X_SOURCE	If a bitmap_ID is specified in the Common block, this field is the X offset into the line style bitmap; otherwise, it is ignored.
DOP_MOVE_R$W_Y_SOURCE	If a bitmap_ID is specified in the Common block, this field is the Y offset into the line style bitmap; otherwise, it is ignored.
DOP_MOVE_R$W_WIDTH	If a bitmap_ID is specified in the Common block, this field is the width of the line style bitmap to be used; otherwise, it is ignored.
DOP_MOVE_R$W_HEIGHT	If a bitmap_ID is specified in the Common block, this field is the height of the line style bitmap to be used; otherwise, it is ignored.
DOP_MOVE_R$W_X_TARGET	The X coordinate of the starting point of the line.
DOP_MOVE_R$W_Y_TARGET	The Y coordinate of the starting point of the line.
DOP_MOVE_R$W_X_TARGET_VEC1	The delta of the X coordinate starting point and end point of vector 1.
DOP_MOVE_R$W_Y_TARGET_VEC1	The delta of the Y coordinate starting point and end point of vector 1.
DOP_MOVE_R$W_L_TARGET_VEC1	This field is irrelevant to this operation.
DOP_MOVE_R$W_X_TARGET_VEC2	The delta of the X coordinate starting point and end point of vector 2.
DOP_MOVE_R$W_Y_TARGET_VEC2	The delta of the Y coordinate starting point and end point of vector 2.
DOP_MOVE_R$W_L_TARGET_VEC2	This field is irrelevant to this operation.

Description

The Draw Complex Line operation permits you to draw a line or patterned line of variable width with a slope in both the length and width direction. See the description of the Draw Line DOP (which also draws lines) to see which routine is appropriate for the operation you want to perform.

Relevant Common Block Fields

You may specify a *bitmap ID* in the Common block to indicate a line style for a complex line. Use the *x_source* and *y_source* fields of the Variable block to specify an offset into the bitmap as a starting point for the line style and use the *width* and *height* fields to specify the extents of the bitmap to use.

By default, the system uses a fill pattern of all 1s.

Initializing the Variable Block

To draw a complex line, you must specify a starting point and two vectors. Vector_1 specifies the length of the line and the slope of the length. Vector_2 specifies the width of the line and the slope of the width. You must specify a delta X and Y value for each vector relative to the target X and Y. To determine the proper X and Y values for the vectors, subtract the X and Y values of the endpoint from those of the starting point to arrive at a delta value.

The length fields are ignored in this operation.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

PERFORMANCE NOTE: Lines are drawn faster if Vector_1 specifies the more gentle slope (less than 45 degrees).

Example

The following FORTRAN program draws a complex line to the screen by taking the following steps:

1. Creating a bitmap pattern.
2. Loading the bitmap, using UISDC$LOAD_BITMAP.
3. Specifying an opcount of 1.
4. Initializing the Variable block with the offset and extent of the specified bitmap pattern.
5. Initializing the Variable block with a starting position of (50,50).
6. Initializing the Variable block with the deltas of the length and width endpoint coordinates.

```
! Calling program
        .
        .
        .
! ************************
! * BITMAP FUNCTION      *
! ************************
```

```
INTEGER*4 FUNCTION GET_BITMAP_ID        ! window ID
! Declare the storage
IMPLICIT INTEGER*4(A-Z)
COMMON /WINDOW/ WD_ID, VD_ID
INTEGER*4 BITMAP_ID
INTEGER*2 BITMAP(16)

! Load the bitmap values
BITMAP(1) = 'AAAA'X
BITMAP(2) = '5555'X
BITMAP(3) = 'AAAA'X
BITMAP(4) = '5555'X
BITMAP(5) = 'AAAA'X
BITMAP(6) = '5555'X
BITMAP(7) = 'AAAA'X
BITMAP(8) = '5555'X
BITMAP(9) = 'AAAA'X
BITMAP(10) = '5555'X
BITMAP(11) = 'AAAA'X
BITMAP(12) = '5555'X
BITMAP(13) = 'AAAA'X
BITMAP(14) = '5555'X
BITMAP(15) = 'AAAA'X
BITMAP(16) = '5555'X

! Load the bitmap from buffer to QDSS memory
BITMAP_ID = UISDC$LOAD_BITMAP (WD_ID,    ! window ID
2                              BITMAP,   ! bitmap address
2                              32,       ! bitmap length (bytes)
2                              16,       ! bitmap width, in pixels
2                              1)        ! bits/pixel

GET_BITMAP_ID = BITMAP_ID
END     ! function

! ***************************
! * COMPLEX LINES SUBROUTINE *
! ***************************

SUBROUTINE SUB_COMPLEX_LINE (DOP, DOP_VAR)
INCLUDE 'VWSSYSDEF'

! Declare the GET_BITMAP_ID function
INTEGER*4 GET_BITMAP_ID

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Associate the predefined variable structure w/ DOP_VAR
RECORD /DOP_MOVE_R_ARRAY/ DOP_VAR
```

Draw Complex Line

```
! Load the values
DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_COMPLEX_LINE
DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_BITMAP_ID = GET_BITMAP_ID()    ! function call

DOP_VAR.DOP_MOVE_R$W_X_SOURCE = 0
DOP_VAR.DOP_MOVE_R$W_Y_SOURCE = 0
DOP_VAR.DOP_MOVE_R$W_WIDTH = 10
DOP_VAR.DOP_MOVE_R$W_HEIGHT = 10
DOP_VAR.DOP_MOVE_R$W_X_TARGET = 50
DOP_VAR.DOP_MOVE_R$W_Y_TARGET = 50
DOP_VAR.DOP_MOVE_R$W_X_TARGET_VEC1 = 300
DOP_VAR.DOP_MOVE_R$W_Y_TARGET_VEC1 = 300
DOP_VAR.DOP_MOVE_R$W_X_TARGET_VEC2 = -20
DOP_VAR.DOP_MOVE_R$W_Y_TARGET_VEC2 = -30

RETURN
END
```

Draw Fixed Text

Describes the additional DOP structure needed to fixed-width text to the screen.

Unique block (text_args)

DOP$W_TEXT_WIDTH	DOP$W_TEXT_HEIGHT
DOP$W_TEXT_STARTING_Y	DOP$W_TEXT_STARTING_X

Field	Use
DOP$W_TEXT_HEIGHT	Height of each character, in pixels. This value is constant for each font.
DOP$W_TEXT_WIDTH	Width of each character.
DOP$W_TEXT_STARTING_X	The viewport relative X coordinate for the starting position of the text on the screen.
DOP$W_TEXT_STARTING_Y	The viewport relative Y coordinate for the starting position of the text on the screen.

Variable block (dop_ftext_array)

DOP_FTEXT$W_OFFSET_Y	DOP_FTEXT$W_OFFSET_X

Field	Use
DOP_FTEXT$W_OFFSET_X	The X coordinate of the offset into the font where a specific character is located.
DOP_FTEXT$W_OFFSET_Y	The Y coordinate of the offset into the font where a specific character is located.

Variable-Length Constant:dop_ftext$c_length

Description

The Draw Fixed Text operation permits you to write fixed-width text to the screen.

To draw scaled, rotated, or differently spaced text, use the Move/Rotate DOP.

Relevant Common Block Fields

To draw fixed-width text to the screen, you must first load a bitmap that contains a fixed-width font from processor memory to the offscreen bitmap memory. You load a bitmap, using the UISDC$LOAD_BITMAP routine. This routine returns a bitmap ID that must be loaded in the *bitmap_ID* field of the Common block. (See Section 5.7 for details about loading bitmaps with the UISDC interface.)

Initializing the Unique Block

The Unique block describes the height and width of the specified font and the position to start writing on the screen. You must initialize all these fields. When writing text, the position you specify as the starting point will be the *upper* left-hand corner of the first character (provided the character height is specified as a positive number). This is unlike specifying *areas*, where the starting point is the *lower* left-hand corner.

Initializing the Variable Block

The Variable block indicates which character to write by specifying an offset into the font.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program writes two characters of text to the screen by taking the following steps:

1. Creating a two-letter bitmap — H I (using a function).
2. Loading the bitmap, using UISDC$LOAD_BITMAP.
3. Specifying an opcount of 2.
4. Initializing the Unique block with a starting position of (100,100).

5. Initializing the Variable block with the offsets of the two characters.

Note that the specified offset values seem to write the letters in reverse order. This is because of the way VAX memory loads the bitmap; the I is loaded at (0,0).

```
! Calling program
        .
        .
        .
! Function that defines
! and loads the bitmap (font)
INTEGER*4 FUNCTION GET_BITMAP_ID        ! window ID

! Declare the storage
IMPLICIT INTEGER*4(A-Z)
COMMON /WINDOW/ WD_ID, VD_ID
INTEGER*4 BITMAP_ID
INTEGER*2 BITMAP(16)

! Load the bitmap values
BITMAP(1) = '423E'X
BITMAP(2) = '4208'X
BITMAP(3) = '4208'X
BITMAP(4) = '4208'X
BITMAP(5) = '7E08'X
BITMAP(6) = '4208'X
BITMAP(7) = '4208'X
BITMAP(8) = '423E'X

! Load the bitmap from buffer to QDSS memory
BITMAP_ID = UISDC$LOAD_BITMAP (WD_ID,    ! window ID
2                              BITMAP,   ! bitmap address
2                              32,       ! bitmap length (bytes)
2                              16,       ! bitmap width, in pixels
2                              1)        ! bits/pixel

GET_BITMAP_ID = BITMAP_ID
END     ! function

! **********
! Subroutine that draws the text
! **********

SUBROUTINE FIXED_TEXT (DOP, DOP_VAR)

IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
! Declare the GET_BITMAP_ID function
INTEGER*4 GET_BITMAP_ID
```

```
! Associate the predefined structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/

    INTEGER*2 OFFSET_X1
    INTEGER*2 OFFSET_Y1

    INTEGER*2 OFFSET_X2
    INTEGER*2 OFFSET_Y2

END STRUCTURE   ! Variable block

! Associate the structure with the DOP
RECORD /VARIABLE_BLOCK/ DOP_VAR

! Load the values
DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_FIXED_TEXT
DOP.DOP$W_OP_COUNT = 2
DOP.DOP$L_BITMAP_ID = GET_BITMAP_ID()        ! function call

DOP.DOP$W_TEXT_HEIGHT = 8
DOP.DOP$W_TEXT_WIDTH = 8
DOP.DOP$W_TEXT_STARTING_X = 100
DOP.DOP$W_TEXT_STARTING_Y = 100

DOP_VAR.OFFSET_X1 = 8   ! reversed in memory
DOP_VAR.OFFSET_Y1 = 0

DOP_VAR.OFFSET_X2 = 0
DOP_VAR.OFFSET_Y2 = 0

RETURN
END
```

Draw Lines

Describes the additional structure needed to draw lines, using specified end points.

Unique block (plot_args)

DOP$W_PLOT_FILL_HEIGHT	DOP$W_PLOT_FILL_WIDTH
DOP$W_PLOT_FILL_PATTERN_Y	DOP$W_PLOT_FILL_PATTERN_X

Field	Use
DOP$W_PLOT_FILL_WIDTH	If a bitmap_ID is specified in the Common block, this field is the width of the line style bitmap. Otherwise, it is ignored.
DOP$W_PLOT_FILL_HEIGHT	If a bitmap_ID is specified in the Common block, this field is the height of the line style bitmap. Otherwise, it is ignored.
DOP$W_PLOT_FILL_PATTERN_X	If a bitmap_ID is specified in the Common block, this field is the X offset into the line style bitmap. Otherwise, it is ignored.
DOP$W_PLOT_FILL_PATTERN_Y	If a bitmap_ID is specified in the Common block, this field is the X offset into the line style bitmap. Otherwise, it is ignored.

Variable block (dop_line_array)

DOP_LINE$W_Y1	DOP_LINE$W_X1
DOP_LINE$W_Y2	DOP_LINE$W_X2

Field	Use
DOP_LINE$W_X1	The X coordinate of the starting point of the line
DOP_LINE$W_Y1	The Y coordinate of the starting point of the line
DOP_LINE$W_X2	The X coordinate of the end point of the line
DOP_LINE$W_Y2	The Y coordinate of the end point of the line

Variable-Length Constant: dop_line$c_length

Description

The Draw Lines operation permits you to draw:

- A line
- A series of lines
- A polygon (a series of connected lines)

Draw Lines differs from Fill Lines in that it begins drawing at the specified starting point, using the bitmap offset specified (if specified). Fill Lines begins drawing at the specified starting point revealing the repeated fill pattern that is relative to the lower left corner of the screen.

The width of lines drawn with this operation is always one pixel. The last pixel of each line is not drawn.

Relevant Common Block Fields

You may specify a bitmap ID in the Common block to indicate a line style to be used when drawing a line. If you do, use the *fill_pattern_x* and *fill_pattern_y* fields of the Unique block to specify an offset into the bitmap as a starting point for the line style and use the *width* and *height* fields to specify the extents of the bitmap to use.

By default, the system uses a fill pattern of all 1s.

Initializing the Unique Block

If you specify a bitmap in the Common block, you must specify the bitmap offset and extents in the Unique block. If you do not specify a bitmap, the Unique block is ignored by a Draw Lines operation.

Initializing the Variable Block

To draw a line, specify the line's two end points in the Variable block of the DOP structure. To draw a series of lines (or a polygon), specify the end points of all the lines in the Variable block and specify the number of lines you wish to draw in the *opcount* field of the Common block.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program takes the following steps to draw a polygon:

1. Uses the predefined structure to initialize the fixed portion of the DOP to write four lines.
2. Defines and initializes the Variable portion of the DOP to hold the end point coordinates of four connecting lines.

```
! Calling program
      .
      .
      .
SUBROUTINE D_LINES (DOP, DOP_VAR)

INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/

    INTEGER*2 FIRST_LINE_X1
    INTEGER*2 FIRST_LINE_Y1
    INTEGER*2 FIRST_LINE_X2
    INTEGER*2 FIRST_LINE_Y2

    INTEGER*2 SECOND_LINE_X1
    INTEGER*2 SECOND_LINE_Y1
    INTEGER*2 SECOND_LINE_X2
    INTEGER*2 SECOND_LINE_Y2
```

```
    INTEGER*2 THIRD_LINE_X1
    INTEGER*2 THIRD_LINE_Y1
    INTEGER*2 THIRD_LINE_X2
    INTEGER*2 THIRD_LINE_Y2

    INTEGER*2 FOURTH_LINE_X1
    INTEGER*2 FOURTH_LINE_Y1
    INTEGER*2 FOURTH_LINE_X2
    INTEGER*2 FOURTH_LINE_Y2

END STRUCTURE   ! dop_structure

! Associate the structure with the DOP_VAR address
RECORD /VARIABLE_BLOCK/ DOP_VAR

! Load the DRAW_LINE values
DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_LINES
DOP.DOP$W_OP_COUNT = 4

DOP_VAR.FIRST_LINE_X1 = 50
DOP_VAR.FIRST_LINE_Y1 = 50
DOP_VAR.FIRST_LINE_X2 = 50
DOP_VAR.FIRST_LINE_Y2 = 75

DOP_VAR.SECOND_LINE_X1 = 50
DOP_VAR.SECOND_LINE_Y1 = 75
DOP_VAR.SECOND_LINE_X2 = 75
DOP_VAR.SECOND_LINE_Y2 = 75

DOP_VAR.THIRD_LINE_X1 = 75
DOP_VAR.THIRD_LINE_Y1 = 75
DOP_VAR.THIRD_LINE_X2 = 75
DOP_VAR.THIRD_LINE_Y2 = 50

DOP_VAR.FOURTH_LINE_X1 = 75
DOP_VAR.FOURTH_LINE_Y1 = 50
DOP_VAR.FOURTH_LINE_X2 = 50
DOP_VAR.FOURTH_LINE_Y2 = 50

RETURN
END
```

Draw Points

Describes the additional DOP structure needed to draw points to the screen.

Unique block (plot_args)

DOP$W_PLOT_FILL_HEIGHT	DOP$W_PLOT_FILL_WIDTH
DOP$W_PLOT_FILL_PATTERN_Y	DOP$W_PLOT_FILL_PATTERN_X

Field	Use
DOP$W_PLOT_FILL_ WIDTH	If a bitmap ID is specified in the Common block, this field is the width of the bitmap. Otherwise, it is ignored.
DOP$W_PLOT_FILL_ HEIGHT	If a bitmap ID is specified in the Common block, this field is the height of the bitmap. Otherwise, it is ignored.
DOP$W_PLOT_FILL_ PATTERN_X	If a bitmap ID is specified in the Common block, this field is the X offset into the bitmap. Otherwise, it is ignored.
DOP$W_PLOT_FILL_ PATTERN_Y	If a bitmap ID is specified in the Common block, this field is the X offset into the bitmap. Otherwise, it is ignored.

Variable Block (dop_point_array)

DOP_POINT$W_Y	DOP_POINT$W_X

Field	Use
DOP_POINT$W_X	The X coordinate of the point to draw
DOP_POINT$W_Y	The Y coordinate of the point to draw

Variable-Length Constant: dop_draw_points$c_length

Description

The Draw Points operation permits you to draw a point, a series of points, or a series of points that correspond to a specified pattern to the screen.

Relevant Common Block Fields

You may specify a bitmap ID in the Common block to indicate a pattern to be used when drawing a point or a series of points. If you do, use the *fill_pattern_x* and *fill_pattern_y* fields of the Variable block to specify an offset into the bitmap as a starting point for the pattern and use the *width* and *height* fields to specify the extents of the bitmap to use.

The pattern is incremented 1 pixel/point and is repeated when the width is reached. This is useful for drawing a thin patterned curve.

By default, the system uses a fill pattern of all 1s.

Initializing the Variable Block

To draw a point, specify the X and Y coordinates of the point in the Variable block. To draw a series of points, specify the X and Y coordinates of all the points in the Variable block and specify the number of points you wish to draw in the *opcount* field of the Common block.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN example draws three points to the screen:

```
! Calling program
        .
        .
        .
SUBROUTINE DRAW_POINT (DOP, DOP_VAR)
INCLUDE 'VWSSYSDEF'
! Associate the predefined structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP
```

```
! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/

    INTEGER*2 FIRST_POINT_X
    INTEGER*2 FIRST_POINT_Y

    INTEGER*2 SECOND_POINT_X
    INTEGER*2 SECOND_POINT_Y

    INTEGER*2 THIRD_POINT_X
    INTEGER*2 THIRD_POINT_Y

END STRUCTURE   ! dop_structure

! Associate the structure with the DOP_VAR address
RECORD /VARIABLE_BLOCK/ DOP_VAR

! Load the DRAW_POINT values
DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_POINTS
DOP.DOP$W_OP_COUNT = 3

DOP_VAR.FIRST_POINT_X = 5
DOP_VAR.FIRST_POINT_Y = 5

DOP_VAR.SECOND_POINT_X = 5
DOP_VAR.SECOND_POINT_Y = 25

DOP_VAR.THIRD_POINT_X = 25
DOP_VAR.THIRD_POINT_Y = 25

RETURN
END
```

Draw Variable Text

Describes the additional DOP structure needed to draw text of variable width to the screen.

Unique Block (text_args)

DOP$W_TEXT_WIDTH	DOP$W_TEXT_HEIGHT
DOP$W_TEXT_STARTING_Y	DOP$W_TEXT_STARTING_X

Field	Use
DOP$W_TEXT_HEIGHT	Height of each character, in pixels; this value is constant for each font
DOP$W_TEXT_WIDTH	Ignored for variable-width fonts
DOP$W_TEXT_STARTING_X	The viewport-relative X coordinate for the starting position of the text on the screen
DOP$W_TEXT_STARTING_Y	The viewport-relative Y coordinate for the starting position of the text on the screen

Variable Block (dop_vtext_array)

DOP_VTEXT$W_OFFSET_Y	DOP_VTEXT$W_OFFSET_X
	DOP_VTEXT$W_WIDTH

Field	Use
DOP_FTEXT$W_OFFSET_X	The X coordinate of the offset into the font where a specific character is located
DOP_FTEXT$W_OFFSET_Y	The Y coordinate of the offset into the font where a specific character is located
DOP_VTEXT$W_WIDTH	Width of the specified character

Variable-Length Constant: dop_vtext$c_length

Description

The Draw Variable Text operation permits you to write variable-width text to the screen.

To draw scaled, rotated, or differently spaced text, use the Move/Rotate DOP.

Relevant Common Block Fields

To draw variable-width text to the screen, you must first load a bitmap that contains a variable-width font from processor memory to the offscreen bitmap memory. You load a bitmap using the UISDC$LOAD_BITMAP routine. This routine returns a bitmap ID that must be loaded in the *bitmap_ID* field of the Common block in order for this operation to succeed. (See Section 5.7 for details about loading bitmaps.)

Initializing the Unique Block

The Unique block describes both the height of the specified font and the position to start writing on the screen. The *width* field is ignored for this operation. Note that when writing text, the position you specify as the starting point is the *upper* left-hand corner of the first character (provided the character height is specified as a positive number). This is unlike specifying an area, where the starting point is the *lower* left-hand corner.

Initializing the Variable Block

The Variable block indicates which character to write by specifying an offset into the font and specifies the width of the character.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program writes three characters of text to the screen (the word LIP) by taking the following steps:

1. Creates a 3-letter bitmap — P L I (using a function).
2. Loads the bitmap, using UISDC$LOAD_BITMAP.
3. Specifies an opcount of 3.
4. Initializes the Unique block with a starting position of (100,100).
5. Initializes the Variable block with the offsets of the three characters (in the proper order) and their respective widths.

Note that the specified offset values seem reversed. This is because of the way VAX memory loads the bitmap; the I is loaded at (0,0).

```
! Calling program
        .
        .
        .
! Function that defines
! and loads the bitmap (font)
INTEGER*4 FUNCTION GET_BITMAP_ID        ! window ID

! Declare the storage
IMPLICIT INTEGER*4(A-Z)
COMMON /WINDOW/ WD_ID, VD_ID
INTEGER*4 BITMAP_ID
INTEGER*2 BITMAP(16)

! Load the bitmap values
BITMAP(1) = '784E'X
BITMAP(2) = '4844'X
BITMAP(3) = '4844'X
BITMAP(4) = '7844'X
BITMAP(5) = '0844'X
BITMAP(6) = '0844'X
BITMAP(7) = '0844'X
BITMAP(8) = '09CE'X

! Load the bitmap from buffer to QDSS memory
BITMAP_ID = UISDC$LOAD_BITMAP (WD_ID,    ! window ID
2                              BITMAP,   ! bitmap address
2                              32,       ! bitmap length (bytes)
2                              16,       ! bitmap width, in pixels
2                              1)        ! bits/pixel
```

```
GET_BITMAP_ID = BITMAP_ID
END     ! function

! **********
! Subroutine that draws the text
! **********

SUBROUTINE VAR_TEXT (DOP, DOP_VAR)

IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
! Declare the GET_BITMAP_ID function
INTEGER*4 GET_BITMAP_ID

! Associate the predefined structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/

    INTEGER*2 OFFSET_X1
    INTEGER*2 OFFSET_Y1
    INTEGER*2 WIDTH1

    INTEGER*2 OFFSET_X2
    INTEGER*2 OFFSET_Y2
    INTEGER*2 WIDTH2

    INTEGER*2 OFFSET_X3
    INTEGER*2 OFFSET_Y3
    INTEGER*2 WIDTH3

END STRUCTURE   ! Variable block

! Associate the structure with the DOP
RECORD /VARIABLE_BLOCK/ DOP_VAR

! Load the values
DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_VAR_TEXT
DOP.DOP$W_OP_COUNT = 3
DOP.DOP$L_BITMAP_ID = GET_BITMAP_ID()        ! function call

DOP.DOP$W_TEXT_HEIGHT = 8
DOP.DOP$W_TEXT_STARTING_X = 100
DOP.DOP$W_TEXT_STARTING_Y = 100

DOP_VAR.OFFSET_X1 = 5   ! reversed in memory
DOP_VAR.OFFSET_Y1 = 0   ! 'L'
DOP_VAR.WIDTH1 = 5

DOP_VAR.OFFSET_X2 = 0   ! 'I'
DOP_VAR.OFFSET_Y2 = 0
DOP_VAR.WIDTH2 = 5
```

```
DOP_VAR.OFFSET_X3 = 10  ! 'P'
DOP_VAR.OFFSET_Y3 = 0
DOP_VAR.WIDTH3 = 6

RETURN
END
```

Fill Lines

Describes the additional structure needed to draw lines, using a specified bitmap pattern.

Unique Block (plot_args)

DOP$W_PLOT_FILL_HEIGHT	DOP$W_PLOT_FILL_WIDTH
DOP$W_PLOT_FILL_PATTERN_Y	DOP$W_PLOT_FILL_PATTERN_X

Field	Use
DOP$W_PLOT_FILL_WIDTH	Width of fill pattern (from offscreen bitmap), in bits; bitmap is identified in *bitmap_ID* field
DOP$W_PLOT_FILL_HEIGHT	Height of fill pattern (from offscreen bitmap), in bits; bitmap is identified in *bitmap_ID* field
DOP$W_PLOT_FILL_PATTERN_X	The X coordinate in the bitmap from which to base the fill pattern; bitmap is identified in *bitmap_ID* field
DOP$W_PLOT_FILL_PATTERN_Y	The Y coordinate in the bitmap from which to base the fill pattern; bitmap is identified in *bitmap_ID* field

Variable Block (dop_line_array)

DOP_LINE$W_Y1	DOP_LINE$W_X1
DOP_LINE$W_Y2	DOP_LINE$W_X2

Field	Use
DOP_LINE$W_X1	The X coordinate of the starting point of the line
DOP_LINE$W_Y1	The Y coordinate of the starting point of the line
DOP_LINE$W_X2	The X coordinate of the end point of the line
DOP_LINE$W_Y2	The Y coordinate of the end point of the line

Variable-Length Constant: dop_line$c_length

Description

The Fill Line operation permits you to draw patterned lines by associating the lines with a bitmap pattern. Fill Lines differs from Draw Lines in that it associates the specified bitmap pattern with the whole screen area and "reveals" the pattern when it draws a line. Draw Lines draws the line, using the specified pattern.

NOTE: This operation is restricted to drawing horizontal lines. This restriction may be lifted in a future release.

Relevant Common Block Fields

To draw a patterned line to the screen, you must first load a bitmap that contains the pattern from processor memory to the offscreen bitmap memory. You load a bitmap, using the UISDC$LOAD_BITMAP routine. This routine returns a bitmap ID that must be loaded in the *bitmap_ID* field of the Common block for this operation to succeed. (See Section 5.7 for details about loading bitmaps.)

Initializing the Unique Block

The Unique block describes which portion of the loaded bitmap pattern should be used to fill the line.

Initializing the Variable Block

To draw a line, specify the line's two end points in the Variable block of the DOP structure. To draw a series of lines, specify the end points of all the lines in the Variable block and specify the number of lines you wish to draw in the *opcount* field of the Common block.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program takes the following steps to draw a filled line:

1. Loads a bitmap pattern using UISDC$LOAD_BITMAP.
2. Specifies DOP$C_FILL_LINES in the *item_type* field.
3. Defines and initializes the Variable portion of the DOP to hold the endpoint coordinates of the line.

```
! Calling program
        .
        .
        .
! Function that defines
! and loads the bitmap
INTEGER*4 FUNCTION GET_BITMAP_ID        ! window ID

! Declare the storage
IMPLICIT INTEGER*4(A-Z)
COMMON /WINDOW/ WD_ID, VD_ID
INTEGER*4 BITMAP_ID
INTEGER*2 BITMAP(16)
! Load the bitmap values
BITMAP(1) = 'AAAA'X
BITMAP(2) = '5555'X
BITMAP(3) = 'AAAA'X
BITMAP(4) = '5555'X
BITMAP(5) = 'AAAA'X
BITMAP(6) = '5555'X
BITMAP(7) = 'AAAA'X
BITMAP(8) = '5555'X
BITMAP(9) = 'AAAA'X
BITMAP(10) = '5555'X
BITMAP(11) = 'AAAA'X
BITMAP(12) = '5555'X
BITMAP(13) = 'AAAA'X
BITMAP(14) = '5555'X
BITMAP(15) = 'AAAA'X
BITMAP(16) = '5555'X
```

```
! Load the bitmap from buffer to QDSS memory
BITMAP_ID = UISDC$LOAD_BITMAP (WD_ID,    ! window ID
2                              BITMAP,   ! bitmap address
2                              32,       ! bitmap length (bytes)
2                              16,       ! bitmap width, in pixels
2                              1)        ! bits/pixel

GET_BITMAP_ID = BITMAP_ID
END     ! function

! **********
! Subroutine that draws the line
! **********

SUBROUTINE F_LINE (DOP, DOP_VAR)

IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
! Declare the GET_BITMAP_ID function
INTEGER*4 GET_BITMAP_ID

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Associate the predefined variable structure w/ DOP_VAR
RECORD /DOP_LINE_ARRAY/ DOP_VAR

! Load the FILL_LINE values
PARAMETER DOP$C_FILL_LINE = 5
DOP.DOP$W_ITEM_TYPE = DOP$C_FILL_LINE

DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_BITMAP_ID = GET_BITMAP_ID()        ! function call

DOP.DOP$W_PLOT_FILL_WIDTH = 16
DOP.DOP$W_PLOT_FILL_HEIGHT = 16
DOP.DOP$W_PLOT_FILL_PATTERN_X = 0
DOP.DOP$W_PLOT_FILL_PATTERN_Y = 0

DOP_VAR.DOP_LINE$W_X1 = 50
DOP_VAR.DOP_LINE$W_Y1 = 50
DOP_VAR.DOP_LINE$W_X2 = 150
DOP_VAR.DOP_LINE$W_Y2 = 50

RETURN
END
```

Fill Point

Describes the additional DOP structure needed to map a point to a defined bitmap.

Unique Block (plot_args)

DOP$W_PLOT_FILL_HEIGHT	DOP$W_PLOT_FILL_WIDTH
DOP$W_PLOT_FILL_PATTERN_Y	DOP$W_PLOT_FILL_PATTERN_X

Field	Use
DOP$W_PLOT_FILL_WIDTH	Width of fill pattern (from offscreen bitmap), in bits; bitmap is identified in *bitmap_ID* field
DOP$W_PLOT_FILL_HEIGHT	Height of fill pattern (from offscreen bitmap), in bits; bitmap is identified in *bitmap_ID* field
DOP$W_PLOT_FILL_PATTERN_X	The X coordinate in the bitmap from which to base the fill pattern; bitmap is identified in *bitmap_ID* field
DOP$W_PLOT_FILL_PATTERN_Y	The Y coordinate in the bitmap from which to base the fill pattern; bitmap is identified in *bitmap_ID* field

Variable Block (dop_point_array)

DOP_POINT$W_Y	DOP_POINT$W_X

Field	Use
DOP_POINT$W_X	The X coordinate of the point to fill
DOP_POINT$W_Y	The Y coordinate of the point to fill

Description

The Fill Point operation permits you to map individual points that you draw to the screen to a bitmap pattern that you specify. If the corresponding bit in the bitmap is set, then the point is drawn to the screen (filled); if not, then the point is not drawn.

Relevant Common Block Fields

To draw patterned points to the screen, you must first load a bitmap that contains the pattern from processor memory to the offscreen bitmap memory. You load a bitmap, using the UISDC$LOAD_BITMAP routine. This routine returns a bitmap ID that must be loaded in the *bitmap_ID* field of the Common block in order for this operation to succeed. (See Section 5.7 for details about loading bitmaps.)

Initializing the Unique Block

The Unique block describes which portion of the loaded bitmap pattern should be used to determine whether to fill the point.

Initializing the Variable Block

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program segment draws 90 points in a patterned sine curve by taking the following steps:

1. Creates a bitmap pattern (in a function).
2. Loads the bitmap, using UISDC$LOAD_BITMAP.
3. Specifies an opcount of 90.
4. Initializes the Unique block.
5. Uses the SIND and REAL functions to calculate the points on the sine curve.
6. Initializes the Variable block with the points by indexing into the Variable block array.

```
! Calling program
        .
        .
        .
! Function that defines
! and loads the bitmap
INTEGER*4 FUNCTION GET_BITMAP_ID        ! window ID

! Declare the storage
IMPLICIT INTEGER*4(A-Z)
COMMON /WINDOW/ WD_ID, VD_ID
INTEGER*4 BITMAP_ID
INTEGER*2 BITMAP(16)

! Load the bitmap values
BITMAP(1) = 'AAAA'X
BITMAP(2) = '5555'X
BITMAP(3) = 'AAAA'X
BITMAP(4) = '5555'X
BITMAP(5) = 'AAAA'X
BITMAP(6) = '5555'X
BITMAP(7) = 'AAAA'X
BITMAP(8) = '5555'X
BITMAP(9) = 'AAAA'X
BITMAP(10) = '5555'X
BITMAP(11) = 'AAAA'X
BITMAP(12) = '5555'X
BITMAP(13) = 'AAAA'X
BITMAP(14) = '5555'X
BITMAP(15) = 'AAAA'X
BITMAP(16) = '5555'X

! Load the bitmap from buffer to QDSS memory
BITMAP_ID = UISDC$LOAD_BITMAP (WD_ID,    ! window ID
2                              BITMAP,   ! bitmap address
2                              32,       ! bitmap length (bytes)
2                              16,       ! bitmap width, in pixels
2                              1)        ! bits/pixel

GET_BITMAP_ID = BITMAP_ID
END     ! function

! ************************
! * FILL POINT SUBROUTINE *
! ************************

SUBROUTINE F_POINT (DOP, DOP_VAR)
```

```
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
! Declare the GET_BITMAP_ID function
INTEGER*4 GET_BITMAP_ID

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/
    INTEGER*2 POINTS(60)
END STRUCTURE   ! dop_structure
! Associate the structure with the DOP_VAR address
RECORD /VARIABLE_BLOCK/ DOP_VAR

! Load the DRAW_POINT values
DOP.DOP$W_ITEM_TYPE = DOP$C_FILL_POINT
DOP.DOP$W_OP_COUNT = 90
DOP.DOP$L_BITMAP_ID = GET_BITMAP_ID()   ! function call
DOP.DOP$W_PLOT_FILL_WIDTH = 16
DOP.DOP$W_PLOT_FILL_HEIGHT = 16
DOP.DOP$W_PLOT_FILL_PATTERN_X = 0
DOP.DOP$W_PLOT_FILL_PATTERN_Y = 0

! Use a loop to load 30 points
! set counters
X = 1
X_COORD = 1
Y_COORD = 2

DO WHILE (X .LE. 91)
   DOP_VAR.POINTS(X_COORD) = X
   DOP_VAR.POINTS(Y_COORD) = SIND(REAL(X)) * 100

   ! Increment counters
   X = X + 1
   X_COORD = X_COORD + 2
   Y_COORD = Y_COORD + 2
END DO

RETURN
END
```

Fill Polygon

Describes the additional DOP structure to create a polygon and fill it with a specified pattern.

Unique block (plot_args)

DOP$W_PLOT_FILL_HEIGHT	DOP$W_PLOT_FILL_WIDTH
DOP$W_PLOT_FILL_PATTERN_Y	DOP$W_PLOT_FILL_PATTERN_X

Field	Use
DOP$W_PLOT_FILL_WIDTH	Width of fill pattern (from offscreen bitmap), in bits; bitmap is identified in *bitmap_ID* field. If no bitmap is specified, this field is ignored.
DOP$W_PLOT_FILL_HEIGHT	Height of fill pattern (from offscreen bitmap), in bits; bitmap is identified in *bitmap_ID* field. If no bitmap is specified, this field is ignored.
DOP$W_PLOT_FILL_PATTERN_X	The X coordinate in the bitmap from which to base the fill pattern; bitmap is identified in *bitmap_ID* field. If no bitmap is specified, this field is ignored.
DOP$W_PLOT_FILL_PATTERN_Y	The Y coordinate in the bitmap from which to base the fill pattern; bitmap is identified in *bitmap_ID* field. If no bitmap is specified, this field is ignored.

Variable Block (dop_poly_array)

DOP_POLY$W_LEFT_Y1	DOP_POLY$W_LEFT_X1
DOP_POLY$W_LEFT_Y2	DOP_POLY$W_LEFT_X2
DOP_POLY$W_RIGHT_Y1	DOP_POLY$W_RIGHT_X1

DOP_POLY$W_RIGHT_Y2	DOP_POLY$W_RIGHT_X2

Field	Use
DOP_POLY$W_LEFT_X1	The X coordinate of the starting point of a line that defines the left edge of a polygon
DOP_POLY$W_LEFT_Y1	The Y coordinate of the starting point of a line that defines the left edge of a polygon
DOP_POLY$W_LEFT_X2	The X coordinate of the end point of a line that defines the left edge of a polygon
DOP_POLY$W_LEFT_Y2	The Y coordinate of the end point of a line that defines the left edge of a polygon
DOP_POLY$W_RIGHT_X1	The X coordinate of the starting point of a line that defines the right edge of a polygon
DOP_POLY$W_RIGHT_Y1	The Y coordinate of the starting point of a line that defines the right edge of a polygon
DOP_POLY$W_RIGHT_X2	The X coordinate of the end point of a line that defines the right edge of a polygon
DOP_POLY$W_RIGHT_Y2	The Y coordinate of the end point of a line that defines the right edge of a polygon

Variable-Length Constant: dop_poly$c_length

Description

The Fill Polygon operation permits you to write a polygon (more precisely, a trapezoid) that is filled with a pattern or a solid color. The trapezoid you create must have top and bottom lines that are both parallel and horizontal. That is, the Y coordinates of the upper corners must be the same, and the Y coordinates of the lower corners must be the same.

NOTE: Some of these restrictions may be lifted in a future release.

Relevant Common Block Fields

To draw a pattern-filled polygon to the screen, you must first load a bitmap that contains the pattern with which you wish to fill the polygon from processor memory to the offscreen bitmap memory. You load a bitmap, using the UISDC$LOAD_BITMAP routine. This routine returns a bitmap ID that must be loaded in the *bitmap_ID* field of the Common block. (See Section 5.7 for details about loading bitmaps.)

To fill the polygon with the solid foreground color, specify a bitmap ID of 0 and omit the Unique block.

Initializing the Unique Block

The Unique block describes which portion of the loaded bitmap pattern should be used to fill the polygon.

Initializing the Variable Block

The Variable block describes the polygon (more precisely, the trapezoid) by specifying the end points of the two lines that are the left and right edges of the polygon. Note that the top and bottom lines of the trapezoid must be both parallel and horizontal. Therefore the Y coordinates of the upper corners must be the same, and the Y coordinates of the lower corners must be the same.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program segment draws a filled polygon by taking the following steps:

1. Creates a bitmap pattern (in a function).
2. Loads the bitmap, using UISDC$LOAD_BITMAP.
3. Specifies an opcount of 1.
4. Initializes the Unique block.
5. Initializes the Variable block with the end points of the two sides of the polygon.

```
! Calling program
        .
        .
        .
! Function that defines
! and loads the bitmap
INTEGER*4 FUNCTION GET_BITMAP_ID        ! window ID

! Declare the storage
IMPLICIT INTEGER*4(A-Z)
COMMON /WINDOW/ WD_ID, VD_ID
INTEGER*4 BITMAP_ID
INTEGER*2 BITMAP(16)
```

Fill Polygon

```
! Load the bitmap values
BITMAP(1) = 'AAAA'X
BITMAP(2) = '5555'X
BITMAP(3) = 'AAAA'X
BITMAP(4) = '5555'X
BITMAP(5) = 'AAAA'X
BITMAP(6) = '5555'X
BITMAP(7) = 'AAAA'X
BITMAP(8) = '5555'X
BITMAP(9) = 'AAAA'X
BITMAP(10) = '5555'X
BITMAP(11) = 'AAAA'X
BITMAP(12) = '5555'X
BITMAP(13) = 'AAAA'X
BITMAP(14) = '5555'X
BITMAP(15) = 'AAAA'X
BITMAP(16) = '5555'X

! Load the bitmap from buffer to QDSS memory
BITMAP_ID = UISDC$LOAD_BITMAP (WD_ID,    ! window ID
2                              BITMAP,   ! bitmap address
2                              32,       ! bitmap length (bytes)
2                              16,       ! bitmap width, in pixels
2                              1)        ! bits/pixel

GET_BITMAP_ID = BITMAP_ID
END     ! function

! **********
! Subroutine that draws the polygon
! **********
SUBROUTINE F_POLYGON (DOP, DOP_VAR)

IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
! Declare the GET_BITMAP_ID function
INTEGER*4 GET_BITMAP_ID

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Associate the predefined variable structure w/ DOP_VAR
RECORD /DOP_POLY_ARRAY/ DOP_VAR

! Load the POLYGON values
DOP.DOP$W_ITEM_TYPE = DOP$C_FILL_POLYGON

DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_BITMAP_ID = GET_BITMAP_ID()        ! function call
```

```
DOP.DOP$W_PLOT_FILL_WIDTH = 16
DOP.DOP$W_PLOT_FILL_HEIGHT = 16
DOP.DOP$W_PLOT_FILL_PATTERN_X = 0
DOP.DOP$W_PLOT_FILL_PATTERN_Y = 0

DOP_VAR.DOP_POLY$W_LEFT_X1 = 10
DOP_VAR.DOP_POLY$W_LEFT_Y1 = 10
DOP_VAR.DOP_POLY$W_LEFT_X2 = 50
DOP_VAR.DOP_POLY$W_LEFT_Y2 = 100

DOP_VAR.DOP_POLY$W_RIGHT_X1 = 150
DOP_VAR.DOP_POLY$W_RIGHT_Y1 = 10
DOP_VAR.DOP_POLY$W_RIGHT_X2 = 100
DOP_VAR.DOP_POLY$W_RIGHT_Y2 = 100

RETURN
END
```

Move Area

Describes the additional DOP structure needed to move (copy) a rectangular area on the screen from one point to another.

Unique Block

The Unique block is not relevant to this operation

Variable Block (dop_move_array)

DOP_MOVE$W_X_SOURCE	DOP_MOVE$W_FILLER
DOP_MOVE$W_WIDTH	DOP_MOVE$W_Y_SOURCE
DOP_MOVE$W_X_TARGET	DOP_MOVE$W_HEIGHT
	DOP_MOVE$W_Y_TARGET

Field	Use
DOP_MOVE$W_FILLER	Reserved for use by DIGITAL
DOP_MOVE$W_X_SOURCE	The X coordinate of the lower left-hand corner of the source area (area to be moved)
DOP_MOVE$W_Y_SOURCE	The Y coordinate of the lower left-hand corner of the source area
DOP_MOVE$W_WIDTH	Width of the area to be moved, in pixels
DOP_MOVE$W_HEIGHT	Height of the area to be moved, in pixels
DOP_MOVE$W_X_TARGET	The X coordinate of the lower left-hand corner of the target area (the area to which the source area is moved)
DOP_MOVE$W_Y_TARGET	The Y coordinate of the lower left-hand corner of the target area

Variable-Length Constant: dop_move$c_length

Description

The Move Area operation permits you to move (copy) a rectangular area from one point to another, either on the screen or in offscreen memory.

Relevant Common Block Fields

The default writing mode that is loaded into the Common block (from the ATB) during allocation is *overlay mode*. When moving areas, you want to use *COPY mode*. To load the UIS$_MODE_COPY value into the *writing_mode* field of the Common block: modify the ATB before allocating using the UIS$SET_WRITING_MODE routine, or load the field with the value directly, using the predefined offsets.

The non-UIS environment writing mode that corresponds to COPY is the WRIT$C_S mode (source only).

Initializing the Variable Block

To move an area, you must specify the coordinates that define the lower left-hand corner of the rectangle you wish to move (source area), the height and width of the area, and the coordinates that define the lower left-hand corner of the area to which you wish to move it (target area).

Example

The following FORTRAN program draws fixed text to the screen at the point (100,100). It then moves (copies) an 8-pixel by 16-pixel rectangle containing the text from (100,100) to (50,50).

In the Move Area DOP, the source rectangle coordinates are (100,93). This is because when text is written to a specified point on the screen (that is, 100,100) the point is considered to be the *upper* left-hand corner of the text — but for Move Area you must specify the *lower* left-hand corner of the rectangle.

The full calling program is shown to clarify the example.

Move Area

```
PROGRAM MOVE_TEXT
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'UISENTRY'
INCLUDE 'UISUSRDEF'
INCLUDE 'VWSSYSDEF'
COMMON /WINDOW/ WD_ID, VD_ID

! Create a display and window
VD_ID = UIS$CREATE_DISPLAY (0.0,0.0,     ! lower left corner
2                           50.0,50.0,   ! upper right corner
2                           15.0,15.0)   ! width & height

WD_ID = UIS$CREATE_WINDOW (VD_ID,                ! display ID
2                          'SYS$WORKSTATION',    ! device name
2                          'DOP Drawing Window') ! window banner

! Allocate the DOP for DRAW_LINES
SIZE = (2 * DOP_FTEXT$C_LENGTH)
DOP1 = UISDC$ALLOCATE_DOP (WD_ID, ! window ID
2                          SIZE,  ! variable portion size, in bytes
2                          0)     ! default ATB number

! Call the FIXED_TEXT subroutine
CALL SUB_FIXED_TEXT (%VAL(DOP1),
2                    %VAL(DOP1+DOP$C_LENGTH))

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,             ! window ID
2                     %VAL(DOP1))        ! DOP address, by value

! Modify the writing mode in ATB
CALL UIS$SET_WRITING_MODE (VD_ID,
2                          0,                    ! default ATB
2                          1,                    ! modified ATB
2                          UIS$C_MODE_COPY)      ! new mode

! Allocate the DOP for MOVE_AREA
SIZE = DOP_MOVE$C_LENGTH
DOP2 = UISDC$ALLOCATE_DOP (WD_ID,        ! window ID
2                          SIZE,         ! size, in bytes
2                          1)            ! number of modified ATB

! Call the MOVE_AREA subroutine
CALL SUB_MOVE_AREA (%VAL(DOP2),
2                   %VAL(DOP2+DOP$C_LENGTH))

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,             ! window ID
2                     %VAL(DOP2))        ! DOP address, by value

CALL SYS$HIBER()

END
```

```
! ************************
! * BITMAP FUNCTION      *
! ************************

INTEGER*4 FUNCTION GET_BITMAP_ID        ! window ID

! Declare the storage
IMPLICIT INTEGER*4(A-Z)
COMMON /WINDOW/ WD_ID, VD_ID
INTEGER*4 BITMAP_ID
INTEGER*2 BITMAP(16)

! Load the bitmap values
BITMAP(1) = '423E'X
BITMAP(2) = '4208'X
BITMAP(3) = '4208'X
BITMAP(4) = '4208'X
BITMAP(5) = '7E08'X
BITMAP(6) = '4208'X
BITMAP(7) = '4208'X
BITMAP(8) = '4208'X

! Load the bitmap from buffer to QDSS memory
BITMAP_ID = UISDC$LOAD_BITMAP (WD_ID,    ! window ID
2                              BITMAP,   ! bitmap address
2                              32,       ! bitmap length (bytes)
2                              16,       ! bitmap width, in pixels
2                              1)        ! bits/pixel

GET_BITMAP_ID = BITMAP_ID
END     ! function

! ******************************
! * DRAW FIXED TEXT SUBROUTINE *
! ******************************

SUBROUTINE SUB_FIXED_TEXT (DOP, DOP_VAR)

IMPLICIT INTEGER*4(A-Z)
INCLUDE 'VWSSYSDEF'
! Declare the GET_BITMAP_ID function
INTEGER*4 GET_BITMAP_ID

! Associate the predefined structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/

    INTEGER*2 OFFSET_X1
    INTEGER*2 OFFSET_Y1
```

```
    INTEGER*2 OFFSET_X2
    INTEGER*2 OFFSET_Y2

END STRUCTURE   ! Variable block

! Associate the Variable block w/ address
RECORD /VARIABLE_BLOCK/ DOP_VAR

! Load the values
DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_FIXED_TEXT
DOP.DOP$W_OP_COUNT = 2
DOP.DOP$L_BITMAP_ID = GET_BITMAP_ID()        ! function call

DOP.DOP$W_TEXT_HEIGHT = 8
DOP.DOP$W_TEXT_WIDTH = 8
DOP.DOP$W_TEXT_STARTING_X = 100
DOP.DOP$W_TEXT_STARTING_Y = 100

DOP_VAR.OFFSET_X1 = 8   ! reversed in memory
DOP_VAR.OFFSET_Y1 = 0

DOP_VAR.OFFSET_X2 = 0
DOP_VAR.OFFSET_Y2 = 0

RETURN
END

! *************************
! * MOVE AREA   SUBROUTINE *
! *************************

SUBROUTINE SUB_MOVE_AREA (DOP,DOP_VAR)

INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Associate predefined Variable Block w/ DOP_VAR
RECORD /DOP_MOVE_ARRAY/ DOP_VAR

! Load the MOVE AREA values
DOP.DOP$W_ITEM_TYPE = DOP$C_MOVE_AREA
DOP.DOP$W_OP_COUNT = 1

DOP_VAR.DOP_MOVE$W_X_SOURCE = 100
DOP_VAR.DOP_MOVE$W_Y_SOURCE = 93        !text is written in
DOP_VAR.DOP_MOVE$W_WIDTH = 16           !negative direction
DOP_VAR.DOP_MOVE$W_HEIGHT = 8
DOP_VAR.DOP_MOVE$W_X_TARGET = 50
DOP_VAR.DOP_MOVE$W_Y_TARGET = 50

RETURN
END
```

Move/Rotate Area

Describes the additional DOP structure to move and rotate an area on the screen to a specified angle (vector) and scale.

Unique Block

The Unique block is not relevant to this operation.

Variable block (dop_move_r_array)

DOP_MOVE_R$W_X_SOURCE	DOP_MOVE_R$W_FILLER
DOP_MOVE_R$W_WIDTH	DOP_MOVE_R$W_Y_SOURCE
DOP_MOVE_R$W_X_TARGET	DOP_MOVE_R$W_HEIGHT
DOP_MOVE_R$W_X_TARGET_VEC1	DOP_MOVE_R$W_Y_TARGET
DOP_MOVE_R$W_L_TARGET_VEC1	DOP_MOVE_R$W_Y_TARGET_VEC1
DOP_MOVE_R$W_Y_TARGET_VEC2	DOP_MOVE_R$W_X_TARGET_VEC2
	DOP_MOVE_R$W_L_TARGET_VEC2

Field	Use
DOP_MOVE_R$W_FILLER	Reserved for use by DIGITAL
DOP_MOVE_R$W_X_SOURCE	The X coordinate of the lower left-hand corner of the source area (area to be moved); if a bitmap_ID is specified in the Common block, this coordinate is relative to the bitmap — otherwise it is viewport relative
DOP_MOVE_R$W_Y_SOURCE	The Y coordinate of the lower left-hand corner of the source area; if a bitmap_ID is specified in the Common block, this coordinate is relative to the bitmap — otherwise it is viewport relative
DOP_MOVE_R$W_WIDTH	Height of the area to be moved, in pixels
DOP_MOVE_R$W_HEIGHT	Width of the area to be moved, in pixels
DOP_MOVE_R$W_X_TARGET	The X coordinate of the lower left-hand corner of the target area (the area to which the source area is moved)
DOP_MOVE_R$W_Y_TARGET	The Y coordinate of the lower left-hand corner of the target area
DOP_MOVE_R$W_X_TARGET_VEC1	The X coordinate of the end point of vector 1 The previously specified corner coordinates are used as the starting point; used to determine the degree of rotation
DOP_MOVE_R$W_Y_TARGET_VEC1	The Y coordinate of the end point of vector 1
DOP_MOVE_R$W_L_TARGET_VEC1	The length of vector 1; determines the degree of scaling
DOP_MOVE_R$W_X_TARGET_VEC2	The X coordinate of the end point of vector 2; the previously specified corner coordinates are used as the starting point; the vector is used to determine the degree of rotation
DOP_MOVE_R$W_Y_TARGET_VEC2	The Y coordinate of the end point of vector 1
DOP_MOVE_R$W_L_TARGET_VEC2	The length of vector 2; determines the degree of scaling

Description

The Move Rotate Area operation permits you to move (copy) a rectangular area from one point to another (either on the screen or in offscreen memory) and permits you to rotate and scale the moved copy.

Move/Rotate Area

This operation also permits you to move/rotate a rectangular area of a bitmap identified by the *bitmap ID* field.

NOTE: If the source is viewport-relative, the viewport must be unobscured and defined as one region for this operation to work properly. This restriction does not apply when the bitmap ID is specified.

NOTE: You must specify a value for all the vector fields even if you desire no scaling, otherwise, your results return undefined.

Relevant Common Block Fields

If you specify the bitmap ID in the *bitmap ID* field of the Common block, the the *x_source* and *y_source* fields of the Variable length block are considered offsets into the bitmap and the area you move/rotate is from the bitmap. A typical use of this feature is to move, rotate, and/or scale text.

The default writing mode that is loaded into the Common block (from the ATB) during allocation is *overlay mode*. When moving areas, you want to use *COPY mode*. To load the UIS$_MODE_COPY value into the *writing mode* field of the Common block, either modify the ATB before allocating using the UIS$SET_WRITING_MODE routine, or load the field with the value directly, using the predefined offsets. The non-UIS environment writing mode that corresponds to COPY is the WRIT$C_S mode (source only).

Initializing the Variable Block

To move an area, you must specify the coordinates that define the lower left-hand corner of the rectangle you wish to move (source area), the height and width of the area, and the coordinates that define the lower left-hand corner of the area to which you wish to move it (target area).

To rotate and scale a moved area, you must specify two vectors. You must specify an X and Y value, as well as a length value for each vector. The X and Y values are used to specify the angle of rotation; the length value specifies the degree of scaling. One vector is related to the X axis the other to the Y axis. To determine the proper X and Y values for the vectors, use the following formulas:

For the X axis vector (VECTOR_1):

```
x = (original width, in pixels) * COS (desired angle of rotation)
y = (original width, in pixels) * SIN (desired angle of rotation)
```

For the Y axis vector (VECTOR_2):

```
x = -(original height, in pixels) * SIN (desired angle of rotation)
y = (original height, in pixels) * COS (desired angle of rotation)
```

The scaling factor is relative to pixels. If the original width of an area is 100 pixels and you specify a VECTOR_1 length of 80, the area will be down-scaled by 20%.

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program draws an 11-pixel by 16-pixel rectangle at the point (50,50). It then moves (copies) the rectangle to the point (150,150) and rotates it 90 degrees. No scaling is specified.

```
PROGRAM MOVE_TEXT
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'UISENTRY'
INCLUDE 'UISUSRDEF'
INCLUDE 'VWSSYSDEF'
INTEGER*4 WD_ID, VD_ID
COMMON /WINDOW/ WD_ID, VD_ID

! Create a display and window
VD_ID = UIS$CREATE_DISPLAY (0.0,0.0,     ! lower left corner
2                           50.0,50.0,   ! upper right corner
2                           15.0,15.0)   ! width & height

WD_ID = UIS$CREATE_WINDOW (VD_ID,                ! display ID
2                          'SYS$WORKSTATION',    ! device name
2                          'DOP Drawing Window') ! window banner

! Allocate the DOP for DRAW_LINES
SIZE = (4 * DOP_LINE$C_LENGTH)
DOP = UISDC$ALLOCATE_DOP (WD_ID, ! window ID
2                         SIZE,  ! variable portion size, in bytes
2                         0)     ! default ATB number

! Call the DRAW LINES subroutine
CALL D_LINES (%VAL(DOP),                ! DOP address, by value
2             %VAL(DOP+DOP$C_LENGTH))   ! Var. block address

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,            ! window ID
2                     %VAL(DOP))        ! DOP address, by value
```

```
! Modify the writing mode in ATB
CALL UIS$SET_WRITING_MODE (VD_ID,
2                          0,                    ! default ATB
2                          1,                    ! modified ATB
2                          UIS$C_MODE_COPY)      ! new mode

! Allocate the DOP for MOVE_ROTATE
SIZE = DOP_MOVE_R$C_LENGTH
DOP1 = UISDC$ALLOCATE_DOP (WD_ID,        ! window ID
2                          SIZE,         ! size, in bytes
2                          1)            ! number of modified ATB

! Call the MOVE_ROTATE subroutine
CALL SUB_MOVE_ROTATE (%VAL(DOP1),
2                     %VAL(DOP1+DOP$C_LENGTH))

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,             ! window ID
2                     %VAL(DOP1))        ! DOP address, by value

CALL SYS$HIBER()

END

! ************************
! * DRAW LINES SUBROUTINE *
! ************************

SUBROUTINE D_LINES (DOP, DOP_VAR)

INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/

    INTEGER*2 FIRST_LINE_X1
    INTEGER*2 FIRST_LINE_Y1
    INTEGER*2 FIRST_LINE_X2
    INTEGER*2 FIRST_LINE_Y2

    INTEGER*2 SECOND_LINE_X1
    INTEGER*2 SECOND_LINE_Y1
    INTEGER*2 SECOND_LINE_X2
    INTEGER*2 SECOND_LINE_Y2

    INTEGER*2 THIRD_LINE_X1
    INTEGER*2 THIRD_LINE_Y1
    INTEGER*2 THIRD_LINE_X2
    INTEGER*2 THIRD_LINE_Y2
```

```
    INTEGER*2 FOURTH_LINE_X1
    INTEGER*2 FOURTH_LINE_Y1
    INTEGER*2 FOURTH_LINE_X2
    INTEGER*2 FOURTH_LINE_Y2

END STRUCTURE   ! dop_structure

! Associate the structure with the DOP_VAR address
RECORD /VARIABLE_BLOCK/ DOP_VAR

! Load the DRAW_LINE values
DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_LINES
DOP.DOP$W_OP_COUNT = 4

DOP_VAR.FIRST_LINE_X1 = 50
DOP_VAR.FIRST_LINE_Y1 = 50
DOP_VAR.FIRST_LINE_X2 = 50
DOP_VAR.FIRST_LINE_Y2 = 65

DOP_VAR.SECOND_LINE_X1 = 50
DOP_VAR.SECOND_LINE_Y1 = 65
DOP_VAR.SECOND_LINE_X2 = 60
DOP_VAR.SECOND_LINE_Y2 = 65

DOP_VAR.THIRD_LINE_X1 = 60
DOP_VAR.THIRD_LINE_Y1 = 65
DOP_VAR.THIRD_LINE_X2 = 60
DOP_VAR.THIRD_LINE_Y2 = 50

DOP_VAR.FOURTH_LINE_X1 = 60
DOP_VAR.FOURTH_LINE_Y1 = 50
DOP_VAR.FOURTH_LINE_X2 = 50
DOP_VAR.FOURTH_LINE_Y2 = 50

RETURN
END

! ***************************
! * MOVE ROTATE   SUBROUTINE *
! ***************************

SUBROUTINE SUB_MOVE_ROTATE (DOP, DOP_VAR)
INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Associate the predefined variable structure w/ DOP_VAR
RECORD /DOP_MOVE_R_ARRAY/ DOP_VAR

! Load the values
DOP.DOP$W_ITEM_TYPE = DOP$C_MOVE_ROTATE_AREA
DOP.DOP$W_OP_COUNT = 1
```

```
DOP_VAR.DOP_MOVE_R$W_X_SOURCE = 50
DOP_VAR.DOP_MOVE_R$W_Y_SOURCE = 50
DOP_VAR.DOP_MOVE_R$W_WIDTH = 11
DOP_VAR.DOP_MOVE_R$W_HEIGHT = 16
DOP_VAR.DOP_MOVE_R$W_X_TARGET = 150
DOP_VAR.DOP_MOVE_R$W_Y_TARGET = 150
DOP_VAR.DOP_MOVE_R$W_X_TARGET_VEC1 = (11 * COSD(90.))
DOP_VAR.DOP_MOVE_R$W_Y_TARGET_VEC1 = (11 * SIND(90.))
DOP_VAR.DOP_MOVE_R$W_L_TARGET_VEC1 = 11
DOP_VAR.DOP_MOVE_R$W_X_TARGET_VEC2 = (-16 * SIND(90.))
DOP_VAR.DOP_MOVE_R$W_Y_TARGET_VEC2 = (16 * COSD(90.))
DOP_VAR.DOP_MOVE_R$W_L_TARGET_VEC2 = 16

RETURN
END
```

Resume Viewport Activity

Describes the additional DOP structure needed for a system viewport to resume activity on a suspended viewport.

Unique Block (stop_args)

DOP$L_DRIVER_VP_ID

Field	Use
DOP$L_DRIVER_VP_ID	The viewport ID associated with the request queue to be resumed

Description

The Resume Viewport Activity operation resumes activity on a viewport that was suspended using the Suspend Viewport Activity DOP (or Suspend Viewport Activity QIO function).

Since the target viewport is suspended, this DOP must be inserted on the queue of a viewport that is not suspended. In most cases, the systemwide viewport is used to resume suspended viewports.

Initializing the Unique Block

The Unique block specifies the viewport ID of the viewport to be resumed. You obtain the viewport ID with the Get Viewport ID QIO at viewport creation time.

Initializing the Variable Block

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program resumes activity on a viewport whose ID is passed to the subroutine:

```
! Calling Program
        .
        .
        .
! ********************
! * RESUME SUBROUTINE *
! ********************
SUBROUTINE RESUME (DOP, DOP_VAR, VP_ID)
INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Load the value
DOP.DOP$W_ITEM_TYPE = DOP$C_RESUME
DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_DRIVER_VP_ID = VP_ID

RETURN
END
```

Scroll Area

Describes the additional DOP structure needed to scroll the screen display.

Unique Block

The Unique block is not relevant to this operation

Variable block (dop_move_array)

DOP_MOVE$W_X_SOURCE	DOP_MOVE$W_FILLER
DOP_MOVE$W_WIDTH	DOP_MOVE$W_Y_SOURCE
DOP_MOVE$W_X_TARGET	DOP_MOVE$W_HEIGHT
	DOP_MOVE$W_Y_TARGET

Field	Use
DOP_MOVE$W_FILLER	Reserved for use by DIGITAL
DOP_MOVE$W_X_SOURCE	The X coordinate of the lower left-hand corner of the source area (area to be moved)
DOP_MOVE$W_Y_SOURCE	The Y coordinate of the lower left-hand corner of the source area
DOP_MOVE$W_WIDTH	Height of the area to be moved, in pixels
DOP_MOVE$W_HEIGHT	Width of the area to be moved, in pixels
DOP_MOVE$W_X_TARGET	The X coordinate of the lower left-hand corner of the target area (the area to which the source area is moved)
DOP_MOVE$W_Y_TARGET	The Y coordinate of the lower left-hand corner of the target area

Variable-Length Constant: dop_move$c_length

Description

The Scroll Area operation permits you to move (copy) a rectangular area either on the screen or in offscreen memory. It differs from the Move Area operation in that it erases (fills with background color) the area specified as the source. This operation is typically used for onscreen scrolling.

Relevant Common Block Fields

Scroll Area always uses the *copy* writing mode. It ignores any value currently contained in the *writing_mode* field.

Initializing the Variable Block

To scroll an area, you must specify the coordinates that define the lower left-hand corner of the rectangle you wish to move (source area), the height and width of the area, and the coordinates that define the lower left-hand corner of the area to which you wish to move it (target area).

Example

The following FORTRAN program scrolls an 11-pixel by 16-pixel rectangle from the point (50,50) to the point (50,150):

```
PROGRAM SCROLL
IMPLICIT INTEGER*4(A-Z)
INCLUDE 'UISENTRY'
INCLUDE 'UISUSRDEF'
INCLUDE 'VWSSYSDEF'
COMMON /WINDOW/ WD_ID, VD_ID

! Create a display and window
VD_ID = UIS$CREATE_DISPLAY (0.0,0.0,     ! lower left corner
2                           50.0,50.0,   ! upper right corner
2                           15.0,15.0)   ! width & height

WD_ID = UIS$CREATE_WINDOW (VD_ID,                ! display ID
2                          'SYS$WORKSTATION',    ! device name
2                          'DOP Drawing Window') ! window banner

! Allocate the DOP for DRAW_LINES
SIZE = (4 * DOP_LINE$C_LENGTH)
DOP = UISDC$ALLOCATE_DOP (WD_ID, ! window ID
2                         SIZE,  ! variable portion size, in bytes
2                         0)     ! default ATB number
```

```
! Call the DRAW LINES subroutine
CALL D_LINES (%VAL(DOP),                ! DOP address, by value
2             %VAL(DOP+DOP$C_LENGTH))   ! Var. block address

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,            ! window ID
2                     %VAL(DOP))        ! DOP address, by value

! Modify the writing mode in ATB
CALL UIS$SET_WRITING_MODE (VD_ID,
2                          0,                    ! default ATB
2                          1,                    ! modified ATB
2                          UIS$C_MODE_COPY)      ! new mode

! Allocate the DOP for SCROLL
SIZE = DOP_MOVE$C_LENGTH
DOP1 = UISDC$ALLOCATE_DOP (WD_ID,       ! window ID
2                          SIZE,        ! size, in bytes
2                          1)           ! number of modified ATB

! Call the MOVE_ROTATE subroutine
CALL SUB_SCROLL (%VAL(DOP1),
2                %VAL(DOP1+DOP$C_LENGTH))

! Queue the DOP asynchronously
CALL UISDC$QUEUE_DOP (WD_ID,            ! window ID
2                     %VAL(DOP1))       ! DOP address, by value

CALL SYS$HIBER()

END

! *************************
! * DRAW LINES SUBROUTINE *
! *************************

SUBROUTINE D_LINES (DOP, DOP_VAR)

INCLUDE 'VWSSYSDEF'

! Associate the predefined structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Build the Variable Block
STRUCTURE /VARIABLE_BLOCK/

    INTEGER*2 FIRST_LINE_X1
    INTEGER*2 FIRST_LINE_Y1
    INTEGER*2 FIRST_LINE_X2
    INTEGER*2 FIRST_LINE_Y2
```

```
        INTEGER*2 SECOND_LINE_X1
        INTEGER*2 SECOND_LINE_Y1
        INTEGER*2 SECOND_LINE_X2
        INTEGER*2 SECOND_LINE_Y2

        INTEGER*2 THIRD_LINE_X1
        INTEGER*2 THIRD_LINE_Y1
        INTEGER*2 THIRD_LINE_X2
        INTEGER*2 THIRD_LINE_Y2

        INTEGER*2 FOURTH_LINE_X1
        INTEGER*2 FOURTH_LINE_Y1
        INTEGER*2 FOURTH_LINE_X2
        INTEGER*2 FOURTH_LINE_Y2
    END STRUCTURE   ! dop_structure

    ! Associate the structure with the DOP_VAR address
    RECORD /VARIABLE_BLOCK/ DOP_VAR

    ! Load the DRAW_LINE values
    DOP.DOP$W_ITEM_TYPE = DOP$C_DRAW_LINES
    DOP.DOP$W_OP_COUNT = 4

    DOP_VAR.FIRST_LINE_X1 = 50
    DOP_VAR.FIRST_LINE_Y1 = 50
    DOP_VAR.FIRST_LINE_X2 = 50
    DOP_VAR.FIRST_LINE_Y2 = 65

    DOP_VAR.SECOND_LINE_X1 = 50
    DOP_VAR.SECOND_LINE_Y1 = 65
    DOP_VAR.SECOND_LINE_X2 = 60
    DOP_VAR.SECOND_LINE_Y2 = 65

    DOP_VAR.THIRD_LINE_X1 = 60
    DOP_VAR.THIRD_LINE_Y1 = 65
    DOP_VAR.THIRD_LINE_X2 = 60
    DOP_VAR.THIRD_LINE_Y2 = 50

    DOP_VAR.FOURTH_LINE_X1 = 60
    DOP_VAR.FOURTH_LINE_Y1 = 50
    DOP_VAR.FOURTH_LINE_X2 = 50
    DOP_VAR.FOURTH_LINE_Y2 = 50

    RETURN
    END

    ! ****************************
    ! * SCROLL        SUBROUTINE *
    ! ****************************

    SUBROUTINE SUB_SCROLL (DOP, DOP_VAR)
    INCLUDE 'VWSSYSDEF'
```

```
! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Associate the predefined variable structure w/ DOP_VAR
RECORD /DOP_MOVE_ARRAY/ DOP_VAR

! Load the values
DOP.DOP$W_ITEM_TYPE = DOP$C_SCROLL_AREA
DOP.DOP$W_OP_COUNT = 1

DOP_VAR.DOP_MOVE$W_X_SOURCE = 50
DOP_VAR.DOP_MOVE$W_Y_SOURCE = 50
DOP_VAR.DOP_MOVE$W_WIDTH = 11
DOP_VAR.DOP_MOVE$W_HEIGHT = 16
DOP_VAR.DOP_MOVE$W_X_TARGET = 50
DOP_VAR.DOP_MOVE$W_Y_TARGET = 150

RETURN
END
```

Start Request Queue

Describes the additional DOP structure needed for the system viewport to restart a *stopped* request queue on another viewport.

Unique Block (stop_args)

DOP$L_DRIVER_VP_ID

Field	Use
DOP$L_DRIVER_VP_ID	The viewport ID associated with the request queue to be started

Description

The Start Request Queue operation starts (or restarts) the processing of packets on the request queue of the specified viewport.

Typically, this call is made after the queue has been stopped with the Stop operation. Since the target viewport is stopped, this DOP must be inserted on the queue of a viewport that is not stopped. In most cases, the systemwide viewport is used to start stopped viewports.

Initializing the Unique Block

The Unique block specifies the viewport ID of the viewport to be started. You obtain the viewport ID with the Get Viewport ID QIO at viewport creation time.

Initializing the Variable Block

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program starts activity on a viewport whose ID is passed to the subroutine:

```
! Calling Program
        .
        .
        .
! **********************
! * START SUBROUTINE *
! **********************
SUBROUTINE START (DOP, DOP_VAR, VP_ID)
INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Load the value
DOP.DOP$W_ITEM_TYPE = DOP$C_START
DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_DRIVER_VP_ID = VP_ID

RETURN
END
```

Stop Request Queue

Describes the additional DOP structure needed to stop removing entries from the specified viewport's request queue.

Unique Block (stop_args)

DOP$L_DRIVER_VP_ID

Field	Use
DOP$L_DRIVER_VP_ID	The viewport ID associated with the request queue to be stopped

Description

The Stop Request Queue operation stops processing on the request queue of the specified viewport to give the calling process control over the viewport bitmap. Stopping a viewport's request queue ensures that no other processes will modify the bitmap of the stopped viewport. To guarantee that all DOPs previously on the queue are processed, insert this DOP on the queue with the Insert DOP QIO or the Execute DOP UISDC routine. The distinction between Stop and Suspend is that Stop waits for any DOPs already processing to complete before returning control.

A viewport management task typically uses this operation to change the position or occlusion of any viewport in the system (using the Set Viewport Region QIO).

Once the Stop operation is invoked, no further commands can be executed from the request queue unless the request queue is explicitly restarted with the Start Request Queue QIO, or the Start Request Queue DOP (from a viewport other than the stopped one).

Initializing the Unique Block

The Unique block specifies the viewport ID of the viewport to be stopped. You obtain the viewport ID with the Get Viewport ID QIO at viewport creation time.

Initializing the Variable Block

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program stops activity on a viewport whose ID is passed to the subroutine:

```
! Calling Program
        .
        .
        .
! ********************
! * STOP SUBROUTINE *
! ********************
SUBROUTINE STOP (DOP, DOP_VAR, VP_ID)
INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Load the value
DOP.DOP$W_ITEM_TYPE = DOP$C_STOP
DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_DRIVER_VP_ID = VP_ID

RETURN
END
```

Suspend Viewport Activity

Describes the additional DOP structure to suspend activity on a viewport.

Unique Block (stop_args)

DOP$L_DRIVER_VP_ID

Field	Use
DOP$L_DRIVER_VP_ID	The viewport ID associated with the request queue to be suspended

Description

The Suspend Viewport Activity operation suspends activity on a specified viewport.

Typically, this operation is used to synchronize drawing operations. When operations are completed, you can resume activity on the viewport by making a call to the Resume QIO, or by invoking the Resume DOP (from a viewport other than the stopped one). Suspend and Resume can be thought of as the drawing parallels to the CTRL/S and CTRL/Q key functions.

The distinction between Stop and Suspend is that Stop waits for any DOPs already processing to complete before returning control.

Initializing the Unique Block

The Unique block specifies the viewport ID of the viewport to be suspended. You obtain the viewport ID with the Get Viewport ID QIO at viewport creation time.

Initializing the Variable Block

The QDSS bitmap-memory coordinate system uses an X coordinate that increases from left to right and a Y coordinate that increases down. For example, to access the top left-hand corner of a bitmap in QDSS memory, specify a 0 for both the X and Y source coordinates.

Example

The following FORTRAN program suspends activity on a viewport whose ID is passed to the subroutine:

```
! Calling Program
      .
      .
      .
! *********************
! * SUSPEND SUBROUTINE *
! *********************
SUBROUTINE SUSPEND (DOP, DOP_VAR, VP_ID)
INCLUDE 'VWSSYSDEF'

! Associate the predefined fixed structure w/ DOP
RECORD /DOP_STRUCTURE/ DOP

! Load the value
DOP.DOP$W_ITEM_TYPE = DOP$C_SUSPEND
DOP.DOP$W_OP_COUNT = 1
DOP.DOP$L_DRIVER_VP_ID = VP_ID

RETURN
END
```

5.7 UISDC DOP Interface

The UISDC DOP interface is provided so that DOPs can be used by applications that draw within UIS viewports. The five UISDC routines that make up the interface permit an application to:

- Allocate storage for a DOP — UISDC$ALLOCATE_DOP.
- Load bitmaps from processor memory into offscreen bitmap memory (for subsequent use by text or fill pattern DOPs) — UISDC$LOAD_BITMAP.
- Submit DOPs to the request queue for execution — UISDC$EXECUTE_DOP_ASYNCH, UISDC$EXECUTE_DOP_SYNCH, and UISDC$QUEUE_DOP.

This section describes each routine.

How to use UISDC$ALLOCATE_DOP to allocate DOPs is described in Section 5.3.1.

How to use UISDC$EXECUTE_DOP_ASYNCH, UISDC$EXECUTE_DOP_SYNCH, and UISDC$QUEUE_DOP to execute DOPs (and how they differ from one another) is described in Section 5.3.2.

How to use UISDC$LOAD_BITMAP is described below.

5.7.1 Loading Bitmaps into Offscreen Memory

To load a bitmap, use the UISDC$LOAD_BITMAP routine.

UISDC$LOAD_BITMAP returns a bitmap identifier (a handle) when you specify a window ID (in which the bitmap will be used), the bitmap address, the length and width of the bitmap, and the number of bits per pixel (more than one on color systems).

There are two ways to use UISDC$LOAD_BITMAP:

- One causes the bitmap to be copied from the user's buffer into a buffer maintained by the driver. When the bitmap is accessed it is copied from the driver-maintained buffer into offscreen memory.
- The other creates a handle (identifier) for the bitmap, but relies on the application to supply the bitmap when it is accessed. This second method saves space by not loading bitmaps until they are actually accessed.

To have the driver maintain the bitmap, specify the address of the bitmap in process memory as the bitmap address parameter. To access the bitmap in a subsequent DOP, load the *bitmap_ID* field of the DOP Common block with the bitmap ID returned by Load Bitmap. The system handles the storage of this bitmap.

When the system manages bitmap storage, it uses the bitmap glyph as a backing store address if it has to swap the bitmap out of offscreen memory. That is, when the bitmap is accessed, the system uses the bitmap glyph to swap the bitmap back into offscreen memory.

To dynamically load a bitmap, specify 0 as the bitmap address parameter, *but still specify the correct length, width, and bits-per-pixel.* To access the bitmap in a subsequent DOP, load the *bitmap_ID* field of the DOP Common block with the bitmap ID returned by LOAD_BITMAP *and* load the *bitmap_glyph* field of the Common block with the address of the bitmap in processor memory.

When you specify a bitmap address of 0 and put the true address in the *bitmap_glyph* field, you save system resources. The bitmap is not loaded until it is accessed, and the application, not the system, is responsible for saving the bitmap (when it is swapped out, it is unknown by the system).

UISDC$ALLOCATE_DOP—Allocate Drawing Packet

Allocates a driver drawing operation primitive (DOP) for a particular display window.

Format

dop_address=**UISDC$ALLOCATE_DOP** *wd_id ,size ,atb*

Returns

VMS Usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword value returned as the address of the DOP in the variable *dop_address* or R0 (VAX MACRO). Used in subsequent execution calls.

UISDC$ALLOCATE_DOP signals all errors; no condition values are returned.

Arguments

wd_id
VMS Usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The window identifier. The **wd_id** argument is the address of a longword that uniquely identifies the display window. This routine associates the DOP with a window by loading the viewport-related fields of the Common block with the coordinates from the specified window (to determine the clipping rectangle). The window identifier is returned to an application at viewport creation time. See UIS$CREATE_WINDOW in the *MicroVMS Workstation Graphics Programming Guide* for more information about the **wd_id** argument.

size
VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Size of the variable portion of the DOP, in bytes. On *input*, the **size** argument is the address of a number that defines the requested size of the variable portion of the DOP to be allocated.

On *output*, it is the size of the variable portion of the DOP that was actually allocated. The size allocated may be smaller than the size you request. Always use the *returned* size in subsequent operations.

atb

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Attribute block number. The **atb** argument is the address of the attribute block which is used to initialize the color, *writing_mode*, and *writing_mask* fields of the Common block.

UISDC$LOAD_BITMAP—Load Bitmap

Loads a bitmap from processor memory into offscreen memory.

Format

bitmap_id=**UISDC$LOAD_BITMAP** *wd_id ,bitmap_adr*
,bitmap_len
,bitmap_width
,bits_per_pixel

Returns

VMS Usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword value returned as the bitmap identifier in the variable *bitmap_id* or R0 (VAX MACRO). This value is used in DOP$L_BITMAP_ID field of subsequent driver DOPs.

UISDC$LOAD_BITMAP signals all errors; no condition values are returned.

Arguments

wd_id
VMS Usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Display window identifier. The **wd_id** argument is the address of a longword that uniquely identifies a display window. The window identifier is returned to an application at viewport creation time. See UIS$CREATE_WINDOW in the *MicroVMS Workstation Graphics Programming Guide* for more information about the **wd_id**.

bitmap_adr
VMS Usage: **address**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Bitmap address. The **bitmap_adr** argument is the address of a bitmap located in processor memory.

bitmap_len

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Bitmap length. The **bitmap_len** argument is the address of the number that defines the length of the bitmap *in bytes*. The length must be a multiple of 2.

bitmap_width

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Width of the bitmap. The **bitmap_width** argument is the address of a number that defines the width of the bitmap *in pixels*. If the width of the bitmap is greater than 1024, the bitmap will wrap. Single-plane bitmaps must have a width that is a multiple of 16.

bits_per_pixel

VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The **bits_per_pixel** argument is the address of a number that defines the number of bits per pixel. Currently, the values *1* and *8* are supported.

Description

UISDC$LOAD_BITMAP loads a bitmap that resides in processor memory into the bitmap portion of offscreen memory. It returns a bitmap ID for the loaded bitmap that can be used in DOPs that require a bitmap ID (fill operations and text operations). In those cases, the returned bitmap ID is loaded into the DOP$C_BITMAP_ID field of the Common block.

UISDC$EXECUTE_DOP_ASYNCH—Execute Drawing Operation Primitive Asynchronously

Starts the execution of the specified drawing operation primitive (DOP) in the specified display window and returns control to the application immediately.

Format

UISDC$EXECUTE_DOP_ASYNCH *wd_id ,dop_address ,iosb*

Returns

UISDC$EXECUTE_DOP_ASYNCH signals all errors; no condition values are returned.

Arguments

wd_id
VMS Usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Display window identifier. The **wd_id** argument is the address of a longword that uniquely identifies the display window. The window identifier is returned to an application at viewport creation time. See UIS$CREATE_WINDOW in the *MicroVMS Workstation Graphics Programming Guide* for more information about the **wd_id** argument.

dop_address
VMS Usage: **vector_byte_unsigned**
type: **byte_unsigned**
access: **read only**
mechanism: **by reference**

DOP. The **dop_address** argument is the address of an array of bytes that comprise the DOP. This address is returned by the UISDC$ALLOCATE_DOP routine.

iosb

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block. The **iosb** argument is the address of an I/O status block that receives a value indicating that the DOP is queued for execution. The IOSB also receives notification when the DOP is completed.

Description

UISDC$EXECUTE_DOP_ASYNCH queues the specified DOP for execution in the specified window's request queue. The execution is performed asynchronously; the DOP is queued for execution, but control is immediately returned to the application. You can use the IOSB to determine when the DOP has actually completed.

UISDC$EXECUTE_DOP_SYNCH—Execute Drawing Operation Primitive Synchronously

Executes the specified drawing operation primitive (DOP) in the specified display window and returns control to the application.

Format

UISDC$EXECUTE_DOP_SYNCH *wd_id ,dop_address*

Returns

UISDC$EXECUTE_DOP_SYNCH signals all errors; no condition values are returned.

Arguments

wd_id
VMS Usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Display window identifier. The **wd_id** argument is the address of a longword that uniquely identifies the display window. The window identifier is returned to an application at viewport creation time. See UIS$CREATE_WINDOW in the *MicroVMS Workstation Graphics Programming Guide* for more information about the **wd_id** argument.

dop_address
VMS Usage: **vector_byte_unsigned**
type: **byte_unsigned**
access: **read only**
mechanism: **by reference**

DOP. The **dop_address** argument is the address of an array of bytes that comprise the DOP. This address is returned by the UISDC$ALLOCATE_DOP routine.

Description

UISDC$EXECUTE_DOP_SYNCH queues the specified DOP for execution on the specified window's request queue. The execution is performed synchronously; the DOP is queued for execution, and EXECUTE_DOP_SYNCH waits until the operation is completed before returning control to the application.

UISDC$QUEUE_DOP—Queue Drawing Operation Primitive

Queues the specified drawing operation primitive for execution in the specified window and returns control to the application.

Format

UISDC$QUEUE_DOP *wd_id ,dop_address*

Returns

UISDC$QUEUE_DOP signals all errors; no condition values are returned.

Arguments

wd_id
VMS Usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Display window identifier. The **wd_id** argument is the address of a longword that uniquely identifies the display window. The window identifier is returned to an application at viewport creation time. See UIS$CREATE_WINDOW for more in the *MicroVMS Workstation Graphics Programming Guide* information about the **wd_id** argument.

dop_address
VMS Usage: **vector_byte_unsigned**
type: **byte_unsigned**
access: **read only**
mechanism: **by reference**

DOP. The **dop_address** argument is the address of an array of bytes that comprise the DOP. This address is returned by the UISDC$ALLOCATE_DOP routine.

Description

UISDC$QUEUE_DOP queues the specified DOP for execution in the specified window's request queue. The execution is performed asynchronously; the DOP is queued for execution, but control is immediately returned to the application. Unlike the UISDC$EXECUTE_DOP_ASYNCH routine, you *cannot* determine when the DOP has actually completed.

Appendix A

QVSS/QDSS Data Types

QVSS/QDSS Data Types

This appendix illustrates and describes the data types common to the QVSS and QDSS systems.

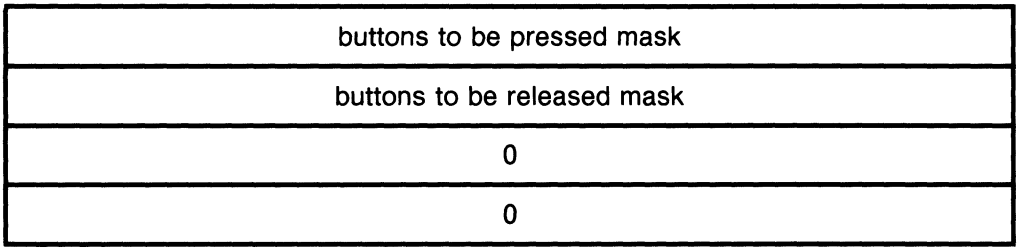
Most of these data types are 2 or 4-longword blocks that must be constructed by your application. However, the more complicated data type, the QVB block is predefined in the SYSLIBRARY:VWSSYSDEF.*lan* definition file — where *lan* is the file extension for the language you are using. (You choose which language definition files are created when you install the system.) The constants that are used in conjunction with the other data types in this appendix are also defined in this file.

Some of the constants that VWSSYSDEF defines are offset values that point to each field in the QVB structure. You can use these predefined offsets in your application to access fields in the QVB.

This appendix labels the QVB with its predefined name and each field in the illustrations with its predefined offset. That is, the predefined QVB name is QVB_COMMON_STRUCTURE and the first field offset is QVB$L_VIDEOSIZE.

You must "include" or "insert" the SYS$LIBRARY:VWSSYSDEF file in each module in which you reference it in order to use any predefined constants and offsets. You should become familiar with the way in which the VWSSYSDEF file defines the DOP structure for the programming language you are using.

Button Simulation Block



Field	Use
buttons to be pressed mask	Mask of the buttons to be pressed.
buttons to be released mask	Mask of the buttons to be released.
	The pointer button definitions used in the masks are defined by the $QVBDEF macro. They consist of the following symbols:

Symbol	Meaning
QV$M_BUTTON_1	Select button
QV$M_BUTTON_2	Button 2
QV$M_BUTTON_3	Button 3
QV$M_BUTTON_4	Button 4

0	This longword must be zero.
0	This longword must be zero.

Cursor Hot Spot

X offset
Y offset

The Pointer Cursor Hot Spot data type defines the pointer cursor hot spot, which is that point within the 16- X 16-pixel cursor display region that is the actual cursor position.

Field	Use
X offset	The X offset from the upper left corner of the pointer pattern to the active point
Y offset	The Y offset from the upper left corner of the pointer pattern to the active point

Data Rectangle Values Block

MINX (left side value)
MINY (bottom side value)
MAXX (right side value)
MAXY (top side value)

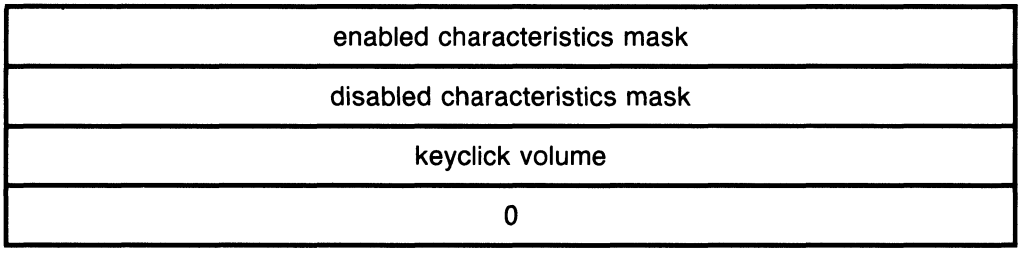
Field	Use
MINX (left side value)	Pixel value for left side of rectangle
MINY (bottom side value)	Pixel value for bottom side of rectangle
MAXX (right side value)	Pixel value for right side of rectangle
MAXY (top side value)	Pixel value for top side of rectangle

Keyboard Request AST Specification Block

AST service routine address
AST parameter
access mode
0

Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST action routine is required. If no AST routine is specified, input will be stored in the typeahead buffer, and delivered either when an AST region is declared, or when an Get Next Input Token QIO is issued.
AST parameter	The user-defined AST parameter is delivered to the AST action routine. It is not examined by the driver.
access mode	The access mode to deliver the AST is maximized with the current access mode.
0	The fourth longword must be zero.

Keyboard Characteristics Block

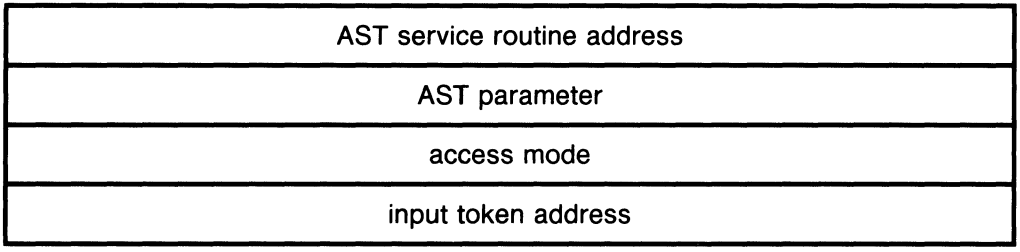


Field	Use
enabled characteristics mask	The first longword is a mask of characteristics to be enabled.
disabled characteristics mask	The second longword is a mask of characteristics to be disabled. The keyboard characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_KEY_AUTORPT	Key held down automatically repeats. Default is on.
QV$M_KEY_KEYCLICK	Keyclick sounds on each keystroke. Default is on.

<table>
<tr><th>Field</th><th>Use</th></tr>
<tr><td></td><td>
<table>
<tr><th>Characteristic</th><th>Meaning</th></tr>
<tr><td>QV$M_KEY_UDF6</td><td>Function keys F6 through F10 generate up/down transitions. Default is off.</td></tr>
<tr><td>QV$M_KEY_UDF11</td><td>Function keys F11 through F14 generate up/down transitions. Default is off.</td></tr>
<tr><td>QV$M_KEY_UDF17</td><td>Function keys F17 through F20 generate up/down transitions. Default is off.</td></tr>
<tr><td>QV$M_KEY_UDHELPDO</td><td>Function keys HELP and DO generate up /down transitions.</td></tr>
<tr><td>QV$M_KEY_UDE1</td><td>Function keys E1 through E6 generate up/down transitions. Default is off.</td></tr>
<tr><td>QV$M_KEY_UDARROW</td><td>Arrow keys generate up/down transitions. Default is off.</td></tr>
<tr><td>QV$M_KEY_UDNUMKEY</td><td>Numeric keypad keys generate up/down transitions. Default is off.</td></tr>
</table>
</td></tr>
<tr><td>keyclick volume</td><td>The keyclick volume must be a value in the range of 1 to 8; 1 is loudest and 8 is softest. If a value of 0 is specified, the current system default keyclick volume is used.</td></tr>
<tr><td>0</td><td>The fourth longword must be 0.</td></tr>
</table>

Keystroke AST Specification Block



Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST action routine is required. If no AST routine is specified, input will be stored in the typeahead buffer, and delivered either when an AST region is declared, or when an Get Next Input Token QIO is issued.
AST parameter	The user-defined AST parameter is delivered to the AST action routine. It is not examined by the driver.
access mode	The access mode to deliver the AST is maximized with the current access mode.
input token address	The input token address is the address of a longword that receives an input token when an AST routine is called. Word 0 of the longword contains token or character data. Values in the range of 0 to 255 map into the DIGITAL multinational character set. Values in the range of 256 to 512 map function keys into token values. Word 1 of the longword contains control information; bit 15 defines the status of a token (1 equals down, 0 equals up). By default an AST is only signaled on a down transition.

Pointer Button Characteristics Block

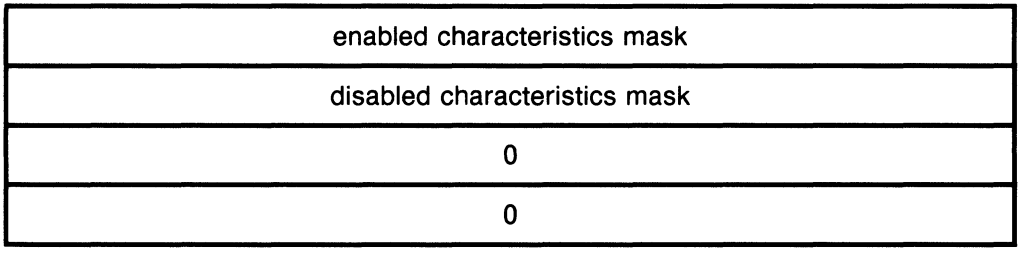
enabled characteristics mask
disabled characteristics mask
0
0

Field	Use
enabled characteristics mask	Longword of characteristics to be enabled.
disabled characteristics mask	Longword of characteristics to be disabled.
	The pointer button characteristics, which are defined by the $QVBDEF macro, consist of the following bit:

Characteristic	Meaning
QV$M_BUT_UPTODOWN	After a pointer button down transition occurs, the current pointer button request receives all future pointer button transitions until all pointer buttons return to the up position (regardless of the position of the pointer cursor on the physical screen). If this characteristic is disabled, then each up and down transition will be delivered to whichever button request is active for the current pointer cursor position. Default is on.

0	This longword must be zero.
0	This longword must be zero.

Pointer Characteristics Block

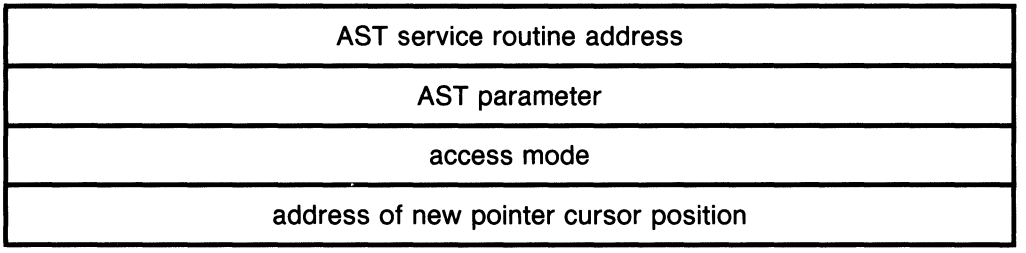


Field	Use
enabled characteristics mask	The first longword is a mask of characteristics to be enabled.
disabled characteristics mask	The second longword is a mask of characteristics to be disabled. The pointer characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_PTR_LEFT_HAND	Invert buttons on mouse or puck. (Buttons 1 and 3 are switched.)
QV$M_PTR_INVERT_STYLUS	Invert buttons on stylus. (Buttons 1 and 3 are switched.)

0	Must be 0.
0	Must be 0.

Pointer Motion AST Specification Block



Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST action routine is required. No buffering of data in the typeahead buffer occurs for pointer motion ASTs.
AST parameter	The user-defined AST parameter is delivered to the AST action routine. It is not examined by the driver.
access mode	The access mode to deliver the AST is maximized with the current access mode.
address of new pointer cursor position	The fourth longword contains the address of a longword to receive the new pointer cursor position when the AST routine is called. (If the information is not required, this longword is 0.) The low-order word contains the new X pixel location of the pointer cursor; the high-order word contains the new Y pixel location of the cursor. For screen pointers, X is in the range 0 through 1023 with the lowest value denoting the left side of the screen; Y is in the range 0 through 863 with the lowest value denoting the bottom of the screen. For tablet pointers, the range of X is defined in the *qvb$w_tablet_width* field of the QVSS block; the range of Y is defined in the *qvb$w_tablet_height* field of the QVSS block.

New Cursor Position

X position on physical screen
Y position on physical screen

Field	Use
X position on physical screen	X position on the physical screen
Y position on physical screen	Y position on the physical screen

New Pointer Position

X position on physical screen
Y position on physical screen

Field	Use
X position on the physical screen	X position on the physical screen
Y position on the physical screen	Y position on the physical screen

QVSS Block (QVB) (qvb_common_structure)

QVB$L_VIDEOSIZE	
QVB$L_VIDEOADDR	
QVB$L_MAPSIZE	
QVB$L_MAPADDR	
QVB$L_CONTEXT	
QVB$L_CSR	
QVB$W_MOUS_YPIX	QVB$W_MOUS_XPIX
QVB$W_HEIGHT	QVB$W_WIDTH
QVB$W_Y_RESOL	QVB$W_X_RESOL
QVB$L_MOUS_XABS	
QVB$L_MOUS_YABS	
QVB$L_MAIN_VIDEOSIZE	
QVB$L_MAIN_VIDEOADDR	
QVB$L_MAIN_MAPSIZE	
QVB$L_MAIN_MAPADDR	
QVB$W_MAIN_MAPMAX	QVB$W_MAIN_MAPMIN
QVB$L_CHARACTERISTICS	
QVB$W_BUTTONS	QVB$W_SCRSAV_TIMOUT
QVB$W_TABLET_XPIX	QVB$W_KEYCLICK_VOLUME
QVB$W_TABLET_WIDTH	QVB$W_TABLET_YPIX
QVB$W_BUT_STATUS	QVB$W_TABLET_HEIGHT

(Data Structure QVSS Block (QVB) cont'd. on next page)

Data Structure: QVSS Block (QVB) (cont'd.)

QVB$W__FLAGS	DEVICE_TYPE	BITS_PER_PIXEL
QVB$W_SPARE_W_1	SPARE_B_1	CURSOR_PLANES
QVB$W_TABLET_YSIZE	QVB$W_TABLET_XSIZE	
QVB$F_TABLET_XRATIO		
QVB$F_TABLET_YRATIO		
QVB$L_UNIT_NUMBER		
QVB$L_POINTER_SETUP		
	QVB$W_HOLD_DEFER_CNT	

The following list describes the contents of each field in the QVSS block.

NOTE: The names contained in the following fields of the preceding data structure have the prefix QVB$B_. The prefixes were omitted in the diagram so that the field names could fit within the fields.

DEVICE_TYPE
BITS_PER_PIXEL
SPARE_B_1
CURSOR_PLANES

Field	Use
QVB$L_VIDEOSIZE	Full size of video memory, in bytes (QVSS specific).
QVB$L_VIDEOADDR	Address of 1st byte of video memory (QVSS specific).
QVB$L_MAPSIZE	Size of scanline map, in words (QVSS specific).
QVB$L_MAPADDR	Address of scanline map (QVSS specific).
QVB$L_CONTEXT	Reserved for use by DIGITAL.
QVB$L_CSR	Reserved for use by DIGITAL.
QVB$W_MOUS_XPIX	X coordinate of the current pointer position.
QVB$W_MOUS_YPIX	Y coordinate of the current pointer position.
QVB$W_WIDTH	Maximum horizontal size of screen, in pixels.
QVB$W_HEIGHT	Maximum vertical size of screen, in pixels.
QVB$W_X_RESOL	Horizontal pixels per inch of physical screen.
QVB$W_Y_RESOL	Vertical pixels per inch of physical screen.

Field	Use
QVB$L_MOUS_XABS	Pointer X position on signed 32-bit virtual coordinate space. Used to obtain relative motion when cursor hardware tracking not used.
QVB$L_MOUS_YABS	Pointer Y position on signed 32-bit virtual coordinate space.
QVB$L_MAIN_VIDEOSIZE	Size of video memory allocated to the windowing system, in bytes (QVSS specific).
QVB$L_MAIN_VIDEOADDR	Address of video memory allocated to the windowing system (QVSS specific).
QVB$L_MAIN_MAPSIZE	Size of the windowing system scanline map (QVSS specific).
QVB$L_MAIN_MAPADDR	Address of the windowing system scanline map (QVSS specific)
QVB$W_MAIN_MAPMIN	Entry number of the lowest entry in the scanline map last updated (QVSS specific).
QVB$W_MAIN_MAPMAX	Entry number of the highest entry in the scanline map last updated (QVSS specific). (Updating these fields permits the driver to update only the modified portion of the scanline map, using the main windowing system scanline map area.)
QVB$L_CHARACTERISTICS	Current systemwide windowing characteristics.
QVB$W_SCRSAV_TIMOUT	Current screen saver timeout value, in seconds.
QVB$W_BUTTONS	This field no longer supported.
QVB$W_KEYCLICK_VOLUME	Default keyclick volume. Value must be in the range of 1 to 8 (1 is loudest). Default is 3.
QVB$W_TABLET_XPIX	The current X position on a tablet, in pixels.
QVB$W_TABLET_YPIX	The current Y position on a tablet, in pixels.
QVB$W_TABLET_WIDTH	Maximum horizontal size of a tablet, in pixels.
QVB$W_TABLET_HEIGHT	Maximum vertical size of a tablet, in pixels.
QVB$W_BUT_STATUS	Current up/down status of the pointing device buttons. If a bit is set, the button is down. The bit positions correspond to the button numbers. That is, the select button, (number 1) is represented by the first bit, and so on.
QVB$B_BITS_PER_PIXEL	Number of bits per pixel. Black and white systems have a value of 1. Color systems may be 4 or 8.
QVB$B_DEVICE_TYPE	Value identifying driver. A value of 0 indicates QVSS, a 1 indicates QDSS.

<table>
<tr><th>Field</th><th>Use</th></tr>
<tr><td>QVB$W_FLAGS</td><td>Internal flags. The following fields are defined within qvb$w_flags:
<table>
<tr><th>Field</th><th>Use</th></tr>
<tr><td>QVB$V_TABLET</td><td>This field is one bit long, and starts at bit 18. Indicates that tablet is present; 0 equals pointer is present.</td></tr>
<tr><td>QVB$V_STYLUS</td><td>This field is one bit long, and starts at bit 19. It indicates that tablet stylus is present.</td></tr>
</table></td></tr>
<tr><td>QVB$B_CURSOR_PLANES</td><td>The number of planes in the hardware cursor. A color cursor has two planes.</td></tr>
<tr><td>QVB$B_SPARE_B_1</td><td>Reserved to DIGITAL.</td></tr>
<tr><td>QVB$W_SPARE_W_1</td><td>Reserved to DIGITAL.</td></tr>
<tr><td>QVB$W_TABLET_XSIZE</td><td>Width of tablet, in centimeters. This is an integer value.</td></tr>
<tr><td>QVB$W_TABLET_YSIZE</td><td>Height of tablet, in centimeters. This is an integer value.</td></tr>
<tr><td>QVB$F_TABLET_XRATIO</td><td>Floating-point ratio of screen width to tablet width, in pixels.</td></tr>
<tr><td>QVB$F_TABLET_YRATIO</td><td>Floating-point ratio of screen height to tablet height, in pixels.</td></tr>
<tr><td>QVB$L_UNIT_NUMBER</td><td>Number of the video device unit associated with the QVB.</td></tr>
<tr><td>QVB$L_POINTER_SETUP</td><td>System characteristics. Contains the current status for system wide pointer characteristics. The following fields are defined within qvb$l_pointer_setup.
<table>
<tr><th>Field</th><th>Use</th></tr>
<tr><td>QV$V_PTR_LEFT_ HAND</td><td>Indicates pointer is left handed if bit is set</td></tr>
<tr><td>QV$V_PTR_INVERT_ STYLUS</td><td>Indicates tip of stylus is select button if bit is set</td></tr>
</table></td></tr>
<tr><td>QVB$W_HOLD_DEFER_CNT</td><td>Reserved for use by DIGITAL.</td></tr>
</table>

Constants - The following constants are defined in conjunction with the QDB$.

Constant	Value
KEY$C_SELECT	The select button.
KEY$C_BUTTON_1	Pointer device button 1.
KEY$C_BUTTON_2	Pointer device button 2.
KEY$C_BUTTON_3	Pointer device button 3.
KEY$C_F1	Function key F1.
KEY$C_F2	Function key F2.
KEY$C_F3	Function key F3.
KEY$C_F4	Function key F4.
KEY$C_F5	Function key F5.
QVB$C_QVSS	QVSS driver type constant (= 0)
QVB$C_QDSS	QDSS driver type constant (= 1)
QVB$C_LENGTH	The length of the QVB structure.

Reserved Function Keystroke AST Specification Block

AST service routine address
AST parameter
access mode
input token address

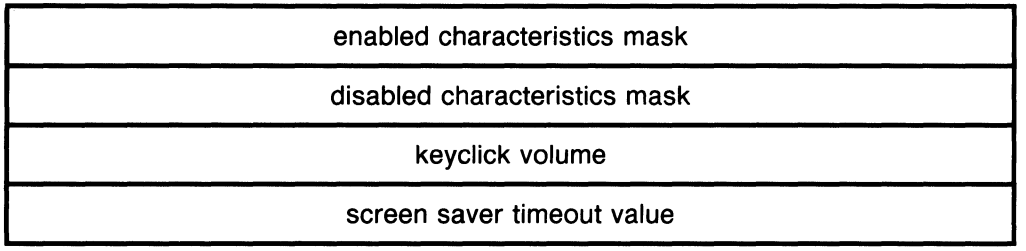
Field	Use
AST service routine address	The address of the AST service routine is 0 if no AST action routine is required.
AST parameter	The user-defined AST parameter is delivered to the AST action routine. It is not examined by the driver.
access mode	The access mode to deliver the AST is maximized with the current access mode.
input token address	The input token address is the address of a longword that receives an input token when an AST routine is called. Word 0 of the longword contains token data defined by the $SMGDEF macro for these function keys. By default an AST is only signaled on a down transition.

Screen Rectangle Values Block

MINX (left side value)
MINY (bottom side value)
MAXX (right side value)
MAXY (top side value)

Field	Use
MINX (left side value)	Pixel value for left side of rectangle
MINY (bottom side value)	Pixel value for bottom side of rectangle
MAXX (right side value)	Pixel value for right side of rectangle
MAXY (top side value)	Pixel value for top side of rectangle

System Characteristics Block



Field	Use
enabled characteristics mask	The first longword is a mask of characteristics to be enabled.
disabled characteristics mask	The second longword is a mask of characteristics to be disabled.

The system characteristics, which are defined by the $QVBDEF macro, consist of the following bits:

Characteristic	Meaning
QV$M_KEY_AUTORPT	Key held down automatically repeats.
QV$M_KEY_KEYCLICK	Keyclick sounds on each keystroke. Default is on.
QV$M_KEY_UDF6	Function keys F6 through F10 generate up/down transitions. Default is off.
QV$M_KEY_UDF11	Function keys F11 through F14 generate up/down transitions. Default is off.
QV$M_KEY_UDF17	Function keys F17 through F20 generate up/down transitions. Default is off.
QV$M_KEY_UDHELPDO	Function keys HELP and DO generate up/down transitions.

Field	Use	
	QV$M_KEY_UDE1	Function keys E1 through E6 generate up/down transitions. Default is off.
	QV$M_KEY_UDARROW	Arrow keys generate up /down transitions. Default is off.
	QV$M_KEY_UDNUMKEY	Numeric keypad keys generate up/down transitions. Default is off.
	QV$M_SYS_SCRSAV	Disable video output to monitor if no input activity occurs within the number of minutes specified in the fourth longword. Any keystroke, pointer button transition, or pointer motion will reset the timer and reactivate a disabled screen. The default is on.
keyclick volume	The keyclick volume must be a value in the range of 1 to 8; 1 is loudest and 8 is softest. Default is 3.	
screen saver timeout value	The screen saver timeout value represents the number of minutes of inactivity that must elapse before the screen saver is activated. The value must be in the range of 1 to 1440. If a value of 0 is specified, then the timeout value is not changed. Default is 15.	

Appendix B

QDSS-Specific Data Types

QDSS-Specific Data Types

This appendix illustrates and describes the data types of the QDSS system.

These data types are predefined in the SYSLIBRARY:VWSSYSDEF.*lan* definition file — where *lan* is the file extension for the language you are using. (You choose which language definition files are created when you install the system.)

VWSSYSDEF defines data type structures and constants, including offset values that define each field in the structure. You can use these predefined offsets in your application to access fields.

This appendix labels each data type with its predefined name and each field in the illustrations with its predefined offset.

For example, VWSSYSDEF defines a structure DOP_STRUCTURE in which it defines an offset DOP$W_ITEM_TYPE. Once you associate the storage with a structure, you can use the offset to reference into the structure.

You must "include" or "insert" the SYS$LIBRARY:VWSSYSDEF file in each module in which you reference it in order to use any predefined constants and offsets. You should become familiar with the way in which the VWSSYSDEF file defines the DOP structure for the programming language you are using.

DOP Queue Structure (req_structure)

REQ$L_REQUEST_FLINK	
REQ$L_REQUEST_BLINK	
REQ$L_RETURN_FLINK	
REQ$L_RETURN_BLINK	
REQ$L_RETURN_LARGE_FLINK	
REQ$L_RETURN_LARGE_BLINK	
REQ$W_LARGE_DOP_SIZE	REQ$W_SMALL_DOP_SIZE
REQ$L_APPLICATION_RESERVED	

The following list describes the contents of each field in the request queue definition.

Field	Use
REQ$L_REQUEST_FLINK	Pending drawing operation queue header
REQ$L_REQUEST_BLINK	Previous drawing operation queue header
REQ$L_RETURN_FLINK	Forward link for the ordinary return queue
REQ$L_RETURN_BLINK	Backward link for the ordinary return queue
REQ$L_RETURN_LARGE_FLINK	Forward link for the large DOP return queue
REQ$L_RETURN_LARGE_BLINK	Backward link for the large DOP return queue
REQ$W_LARGE_DOP_SIZE	Size of a DOP returned to the large DOP return queue
REQ$W_SMALL_DOP_SIZE	Size of a DOP returned to the ordinary return queue
REQ$L_APPLICATION_RESERVED	A longword reserved for use by the application

Constants - The following constants are defined in conjunction with the REQ$.

Constant	Value
REQ$K_RETURN_OFFSET	Offset from beginning of structure to return queue
REQ$K_LENGTH	Length of the structure

QDSS Block (QDB) (qvb_qdss_structure)

QVBDEF$$_QD_COMMON_FILL (120 bytes)	
QDB$L_SYSVP	
QDB$W_ON_SCREEN_Y	QDB$W_ON_SCREEN_X
QDB$W_ON_SCREEN_HEIGHT	QDB$W_ON_SCREEN_WIDTH
QDB$W_SCROLL_Y	QDB$W_SCROLL_X
QDB$W_SCROLL_HEIGHT	QDB$W_SCROLL_WIDTH
QDB$W_FREE_1_Y	QDB$W_FREE_1_X
QDB$W_FREE_1_HEIGHT	QDB$W_FREE_1_WIDTH
QDB$W_FREE_2_Y	QDB$W_FREE_2_X
QDB$W_FREE_2_HEIGHT	QDB$W_FREE_2_WIDTH
QDB$W_FREE_3_Y	QDB$W_FREE_3_X
QDB$W_FREE_3_HEIGHT	QDB$W_FREE_3_WIDTH
QDB$W_FONT_Y	QDB$W_FONT_X
QDB$W_FONT_HEIGHT	QDB$W_FONT_WIDTH

(Data Structure QDSS Block (QDB) cont'd. on next page)

Data Structure: QDSS Block (QDB) (cont'd.)

QDB$W_CLIP_SAVE_Y	QDB$W_CLIP_SAVE_X
QDB$W_CLIP_SAVE_HEIGHT	QDB$W_CLIP_SAVE_WIDTH
QDB$L_COLOR_INDICES	
QDB$L_COLOR_COLORS	
QDB$L_COLOR_RBITS	
QDB$L_COLOR_GBITS	
QDB$L_COLOR_BBITS	
QDB$L_COLOR_IBITS	
QDB$L_COLOR_RES_INDICES	
QDB$L_COLOR_REGEN	
QDB$L_COLOR_MAPS	
QDB$L_COLOR_INTENSITY_FLAG	

The following list describes the contents of each field in the QDSS block. Note that the first part of the QDB is actually the QVB. See Appendix A for a full explanation of the QVB fields.

Field	Use
QVBDEF$_QD_COMMON_FILL	The part of the block occupied by the QVB common block
QDB$L_SYSVP	Systemwide viewport ID
QDB$W_ON_SCREEN_X	X coordinate of the lower left-hand corner of onscreen memory
QDB$W_ON_SCREEN_X	Y coordinate of the lower left-hand corner of onscreen memory
QDB$W_ON_SCREEN_WIDTH	Width of onscreen memory
QDB$W_ON_SCREEN_HEIGHT	Height of onscreen memory
QDB$W_SCROLL_X	X coordinate of the lower left-hand corner of the scroll area
QDB$W_SCROLL_Y	Y coordinate of the lower left-hand corner of the scroll area
QDB$W_SCROLL_WIDTH	Width of the scroll area
QDB$W_SCROLL_HEIGHT	Height of the scroll area
QDB$W_FREE_1_X	X coordinate of the lower left-hand corner of writeable memory
QDB$W_FREE_1_Y	Y coordinate of the lower left-hand corner of writeable memory
QDB$W_FREE_1_WIDTH	Width of writeable memory
QDB$W_FREE_1_HEIGHT	Height of writeable memory
QDB$W_FONT_X	X coordinate of the lower left-hand corner of bitmap storage area
QDB$W_FONT_Y	X coordinate of the lower left-hand corner of bitmap storage area
QDB$W_FONT_WIDTH	Width of the bitmap storage area
QDB$W_FONT_HEIGHT	Height of the bitmap storage area
QDB$L_COLOR_INDICES	Color map size
QDB$L_COLOR_COLORS	Maximum number of possible colors
QDB$L_COLOR_RBITS	Number of bits of precision for red
QDB$L_COLOR_GBITS	Number of bits of precision for green
QDB$L_COLOR_BBITS	Number of bits of precision for blue
QDB$L_COLOR_IBITS	Number of bits of intensity precision
QDB$L_COLOR_RES_INDICES	Number of color map entries reserved by the system
QDB$L_COLOR_REGEN	Color regeneration characteristics; on a QDSS system color regeneration is retroactive
QDB$L_COLOR_MAPS	Always 1 for QDSS

Field	Use
QDB$L_COLOR_INTENSITY_FLAG	Indicates whether color or intensity mode; 1 = system setup for intensity values; 0 = RGB expected

Constants – The following constants are defined in conjunction with the QDB.

Constant	Value
QVB$C_LENGTH	The length of the QVB structure.

Return Queue Structure (ret_structure)

RET$L_RETURN_FLINK	
RET$L_RETURN_BLINK	
RET$L_RETURN_LARGE_FLINK	
RET$L_RETURN_LARGE_BLINK	
RET$W_LARGE_DOP_SIZE	RET$W_SMALL_DOP_SIZE
RET$L_APPLICATION_RESERVED	

The following list describes the contents of each field in the request queue definition.

Field	Use
RET$L_RETURN_FLINK	Forward link for the ordinary return queue
RET$L_RETURN_BLINK	Backward link for the ordinary return queue
RET$L_RETURN_LARGE_FLINK	Forward link for the large DOP return queue
RET$L_RETURN_LARGE_BLINK	Backward link for the large DOP return queue
RET$W_LARGE_DOP_SIZE	Size of a DOP returned to the large DOP return queue
RET$W_SMALL_DOP_SIZE	Size of a DOP returned to the ordinary return queue
RET$L_APPLICATION_RESERVED	A longword reserved for use by the application

Transfer Parameter Block (TPB) (tpb_structure)

TPB$W_X_SOURCE	TPB$B_SIZE	TPB$B_TYPE
TPB$W_WIDTH	TPB$W_Y_SOURCE	
TPB$W_X_TARGET	TPB$W_HEIGHT	
TPB$W_X_TARGET_VEC1	TPB$W_Y_TARGET	
TPB$W_L_TARGET_VEC1	TPB$W_Y_TARGET_VEC1	
TPB$W_Y_TARGET_VEC2	TPB$W_X_TARGET_VEC2	
	TPB$W_L_TARGET_VEC2	

The following list describes the contents of each field in the transfer parameter block.

Field	Use
TPB$B_TYPE	Type of transfer being performed, either bitmap-to-processor (BTP), processor-to-bitmap (PTB), or bitmap-to-bitmap Use the constants defined below to load this field.
TPB$B_SIZE	Reserved to DIGITAL.
TPB$W_X_SOURCE	X coordinate of lower left-hand corner of source bitmap.
TPB$W_Y_SOURCE	Y coordinate of lower left-hand corner of source bitmap.
TPB$W_WIDTH	Width of source bitmap.
TPB$W_HEIGHT	Height of source bitmap.
TPB$W_X_TARGET	X coordinate of lower left-hand corner of target bitmap. (Only specified for bitmap-to-bitmap transfer.)
TPB$W_Y_TARGET	Y coordinate of lower left-hand corner of target bitmap.
TPB$W_X_TARGET_VEC1	Reserved to DIGITAL.
TPB$W_Y_TARGET_VEC1	Reserved to DIGITAL.
TPB$W_L_TARGET_VEC1	Reserved to DIGITAL.
TPB$W_X_TARGET_VEC2	Reserved to DIGITAL.
TPB$W_Y_TARGET_VEC2	Reserved to DIGITAL.
TPB$W_L_TARGET_VEC2	Reserved to DIGITAL.

Constants - The following constants are defined in conjunction with the TPB.

Constant	Value
TPB$C_BITMAP_XFR	Used to specify a bitmap-to-bitmap transfer.
TPB$C_SOURCE_ONLY	Used to specify a BTP or PTB transfer.
TPB$C_SOURCE_LENGTH	Structure length for BTP or PTB transfers
TPB$C_BITMAP_XFR_LENGTH	Structure length for a bitmap-to-bitmap transfer.
TPB$C_LENGTH	Full structure length.

Update Region Definition Block (urd_structure)

URD$W_Y_MIN	URD$W_X_MIN
URD$W_Y_MAX	URD$W_X_MAX
URD$W_Y_BASE	URD$W_X_BASE

The following list describes the contents of each field in the update region definition block.

Field	Use
URD$W_X_MIN	Viewport-relative X coordinate of the lower left-hand corner of defined region
URD$W_Y_MIN	Viewport-relative Y coordinate of the lower left-hand corner of defined region
URD$W_X_MAX	Viewport-relative X coordinate of the upper right-hand corner of defined region
URD$W_Y_MAX	Viewport-relative Y coordinate of the upper right-hand corner of defined region
URD$W_X_BASE	Absolute X coordinate from lower left-hand corner of defined region
URD$W_Y_BASE	Absolute X coordinate from lower left-hand corner of defined region

Constants - The following constants are defined in conjunction with the URD.

Constant	Value
URD$C_LENGTH	Length of the structure.

Appendix C

QDSS Writing Modes

This appendix contains a table of all the QDSS writing modes. They describe the logical operations that may be performed when two graphic objects intersect. This operation occurs between the *destination,* the bits making up the existing bitmap pixel, and the *source index,* a value selected by the application in the DOP structure. The results of this operation are tested against the foreground and background color to determine the bit settings for the pixels making up the intersecting area.

These writing mode names are acronyms that reflect the function being performed by the mode. The names can be interpreted (and remembered) by reading D for destination, S for source index, N for NEGATE, A for AND, O for OR, and X for XOR — with all expressions written in reverse Polish notation.

Table C-1 QDSS Writing Modes

QDSS Writing Modes	Function
WRIT$C_ZEROES	All resulting bits are set to 0.
WRIT$C_DSON	The destination is ORed with source index, then the result is negated.
WRIT$C_DNSA	The destination is negated, then ANDed with the source index.
WRIT$C_DN	The destination is negated.
WRIT$C_DSNA	The source index is negated, then ANDed with the destination.
WRIT$C_SN	The source index is negated.
WRIT$C_DSX	The destination is XORed with the source index.
WRIT$C_DSAN	The destination is ANDed with the source index, then the result is negated.
WRIT$C_DSA	The destination is ANDed with the source index.
WRIT$C_DSXN	The destination is XORed with the source index, then the result is negated.
WRIT$C_S	The result is equal to the source index.
WRIT$C_DNSO	The destination is negated, then ANDed with the source index.
WRIT$C_D	The result is equal to the destination.
WRIT$C_DSNO	The source index is negated, then ORed with the destination.
WRIT$C_DSO	The destination is ORed with the source index.
WRIT$C_ONES	All resulting bits are set to 1.

In addition, you may specify the following modifiers with the writing modes (by ANDing the two values).

Table C-2 QDSS Writing Mode Modifiers

Modifier	Function
WRIT$M_USE_MASK_2	Specifies that the mask specified in the bitmap ID field of the DOP be used to determine whether the resulting pixel should be written.
WRIT$M_COMP_MASK_2	Specifies that the complement of the mask specified in the bitmap ID field of the DOP be used to determine whether the resulting pixel should be written.
WRIT$M_NO_SRC_COMP	Specifies that the complement of the source index should not be used in the logical operation.

Appendix D

QVSS Programming Example

D.1 Programming

This appendix contains a sample application program for the QVSS driver.

D.1.1 Program Functions

The test program in Section 2.1.2 is an example of how a typical program might be designed for the QVSS driver. This simple program, using most of the QIO functions available to the QVSS driver, performs the following operations:

1. Sets system windowing characteristics.
   a. Enables autorepeat and keyclick; disables keys F6 to F10 and the arrow keys from generating up-transition ASTs.
   b. Sounds the bell to indicate that the characteristics have been set.
2. Sets up permanent cursor pattern.
   a. Sets up a new default system cursor pattern.
3. Sets cursor regions — Sets up two separate cursor regions on the screen. One region will be located in the lower left corner and have a block-shaped cursor. The other region will be in the upper right corner and will have a cross-shaped cursor.
4. Sets up two one-shot requests on keyboard channel 1.
5. Sets up keyboard region 1 to be a French keyboard.
6. Sets up a private two-stroke compose table for keyboard region 1.
7. Sets up a private three-stroke compose table for keyboard region 1.
8. Sets keyboard regions.
   a. Sets up two keyboard regions. Each region specifies a keyboard AST and a control AST.

b. The keyboard AST specified for each region performs the following actions:
   - Sends an acknowledgment message to the terminal.
   - Sends (echoes) the character typed to the terminal.
   - If the character was a "C", issues a cycle QIO on the keyboard list.
   - The cycle QIO causes the other keyboard region to become active.
   - A control AST for the new region is then delivered.
   - If the character "F" was typed, sets event flag 2, which in turn terminates the program.

c. The control AST specified for each region sends a message to the terminal indicating that a control AST was delivered.

9. Enables AST region for pointer buttons.
   a. Sets up an AST to be called each time a pointer button is depressed/released in the specified region.
   b. The AST routine determines which button was changed and prints a message identifying the pointer button. The AST routine then determines whether the button is currently up or down and prints a message to that effect.
10. Enables AST for function key F5 — Sets up an AST to be called each time function key F5 is pressed.
    - The F5 AST routine will issue a cycle QIO on the keyboard list.
    - The cycle QIO causes the control AST for the new keyboard region to be delivered.
11. Enables an AST region for pointer motion.
    a. Sets up an AST to be called each time the pointer moves within a specified region.
    b. The AST routine will print a message indicating that the pointer has moved.
12. Simulates keyboard input on keyboard channel 2 — Simulates the input of a character string on keyboard region 2.
13. Waits for event flag 2 to be set.
14. Exits program when event flag 2 is set.

D.1.2 QVSS Program Example

```
        .TITLE  QVSS PORT DRIVER TEST PROGRAM
        .IDENT  /02/

; ************************************************************************
;
;                   QVSS PORT DRIVER TEST PROGRAM
;
; ************************************************************************

        .SBTTL  DECLARATIONS
;
; Define symbols
;

        $IODEF                  ; Define I/O function codes
        $QVBDEF                 ; QVSS definitions

;
; Allocate workstation descriptor and channel number storage
;
WS_DEVNAM:
        .ASCID  /SYS$WORKSTATION/     ; Logical name of workstation
SYS_CHAN1:
        .BLKW   1               ; Channel number storage
CUR_CHAN1:
        .BLKW   1               ; Channel number storage
CUR_CHAN2:
        .BLKW   1               ; Channel number storage
KBD_CHAN1:
        .BLKW   1               ; Channel number storage
KBD_CHAN2:
        .BLKW   1               ; Channel number storage
BUT_CHAN:
        .BLKW   1               ; Channel number storage
MOUSE_CHAN:
        .BLKW   1               ; Channel number storage
FNKEY_F5_CHAN:
        .BLKW   1               ; Channel number storage

BUTTON:                         ; State of buttons
        .BLKL   1
MOUSE_XY:                       ; Current pointer X,Y coordinates
        .BLKL   1

CHARACTER:                      ; Keyboard character
        .BLKL   1
DESC:   .LONG   2
        .LONG   CHARACTER

IOSB_BLOCK:                     ; IOSB descriptor
        .QUAD   0
DESC1:  .LONG   1
        .LONG   IOSB_BLOCK+4
```

D–4 QVSS Programming Example
Programming

```
        .SBTTL  START - MAIN ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
; ******************************************************************
;
;                       START PROGRAM
;
; ******************************************************************
;
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
START:
        .WORD                           ; Entry mask
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; Assign channel using
                   CHAN=SYS_CHAN1       ;  logical name and channel number
        BLBS    R0,5$                   ; Check for success
        BRW     ERROR                   ; Report error on failure

5$:     BSBW    SET_CHARACTERISTICS     ; Set up system characteristics
        BSBW    SET_PERM_CURSOR         ; Set up new system wide cursor pattern
        BSBW    SET_CURSOR              ; Set up cursors
        BSBW    SET_ONESHOT             ; Set up one-shot on keyboard channel 1
        BSBW    SET_FRENCH_KB           ; Set up French keyboard on keyboard 1
        BSBW    SET_COMPOSE2_TABLE      ; Set up 2-stroke compose table on kbd 1
        BSBW    SET_COMPOSE3_TABLE      ; Set up 3-stroke compose table on kbd 1
        BSBW    SET_KBDAST              ; Set keyboard AST
        BSBW    SET_BUTTONAST           ; Set up button region AST
        BSBW    SET_FNKEYAST            ; Set up function key AST
        BSBW    SET_MOUSEAST            ; Set up pointer region AST
        BSBW    SIMULATE_INPUT          ; Simulate input on keyboard 2
        $CLREF_S        EFN=#2          ; Clear event flag #2
        $WAITFR_S       EFN=#2          ; Wait for event flag #2

ERROR:  $EXIT_S R0
```

```
        .SBTTL  SET_CHARACTERISTICS - SET SYSTEM WINDOWING CHARACTERISTICS
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Set a couple of SYSTEM windowing characteristics and sound the
;       bell after they are set.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_CHARACTERISTICS:
        MOVL    #IO$C_QV_MODIFYSYS,R0   ; Modify characteristics code
        $QIOW_S CHAN=SYS_CHAN1,-        ; Use system channel
                FUNC=#IO$_SETMODE,-     ; QIO function code
                P1 = (R0),-             ; QVSS function code
                P4 = #CHAR_BLOCK        ; Characteristics block
        BLBS    R0,10$                  ; Check for success
        BRW     ERROR

10$:    MOVL    #IO$C_QV_SOUND,R0       ; Sound code
        $QIOW_S CHAN=SYS_CHAN1,-
                FUNC=#IO$_SETMODE,-
                P1 = (R0),-
                P2 = #QV$M_SOUND_BELL   ; Bell sound modifier
        BLBS    R0,20$                  ; Check for success
        BRW     ERROR                   ; Report error on failure

20$:    RSB

CHAR_BLOCK:                             ; Characteristics block
        .LONG   <QV$M_SYS_AUTORPT!QV$M_SYS_KEYCLICK>    ; Enable these
        .LONG   <QV$M_SYS_UDF6!QV$M_SYS_UDARROW>        ; Disable these
        .LONG   5                                       ; Keyclick volume
        .LONG   30                                      ; Screen saver timeout
```

```
        .SBTTL  SET_PERM_CURSOR - SET UP NEW SYSTEM CURSOR
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Set a new system wide cursor pattern.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_PERM_CURSOR:
        MOVL    #<IO$C_QV_SETCURSOR -   ; Define system cursor
                !IO$M_QV_LOAD_DEFAULT>,R0 ; Default modifier
        $QIOW_S CHAN=SYS_CHAN1,-
                FUNC=#IO$_SETMODE,-
                P1 =(R0),-
                P2=#MOUSE$BM,-          ; Cursor description
                P4=#MOUSE$HOTSPOT       ; Cursor hot spot
        BLBS    R0,10$                  ; Check for success
        BRW     ERROR                   ; Report error on failure
10$:    RSB

MOUSE$BM:                               ; Bitmap definition of
        .WORD   ^b0000001110000000      ; pointer cursor used in call
        .WORD   ^b0000011000000000
        .WORD   ^b0000001100000000
        .WORD   ^b0000000110000000
        .WORD   ^b0011111111111100
        .WORD   ^b0100000000000010
        .WORD   ^b1101100110011011
        .WORD   ^b1101100110011011
        .WORD   ^b1101100110011011
        .WORD   ^b1100000000000011
        .WORD   ^b1100000000000011
        .WORD   ^b1111000000001111
        .WORD   ^b1100110000110011
        .WORD   ^b0110001111000110
        .WORD   ^b0011010000101100
        .WORD   ^b0001111111111000

MOUSE$HOTSPOT:                          ; Pointer hot spot definition
        .LONG   9
        .LONG   0

; The following two cursor patterns (although not used by the appliation)
; are provided to show alternative patterns
```

```
PENCIL: .WORD   ^X0000                  ; Pencil cursor definition
        .WORD   ^X0000
        .WORD   ^X0700
        .WORD   ^X0880
        .WORD   ^X0880
        .WORD   ^X1700
        .WORD   ^X1100
        .WORD   ^X2200
        .WORD   ^X2200
        .WORD   ^X4400
        .WORD   ^X4400
        .WORD   ^Xc800
        .WORD   ^Xf000
        .WORD   ^Xe000
        .WORD   ^Xc000
        .WORD   ^X8000

PENCIL_HS:                              ; Pencil hot spot definition
        .LONG 0
        .LONG 15

SPRAYCAN:                               ; Spraycan cursor definition
        .WORD   ^X8000
        .WORD   ^X2000
        .WORD   ^X8b00
        .WORD   ^X2780
        .WORD   ^X8580
        .WORD   ^X0fc0
        .WORD   ^X0840
        .WORD   ^X09c0
        .WORD   ^X0940
        .WORD   ^X09c0
        .WORD   ^X0940
        .WORD   ^X09c0
        .WORD   ^X09c0
        .WORD   ^X0840
        .WORD   ^X0840
        .WORD   ^X0fc0

SPRAYCAN_HS:                            ; Spraycan hot spot definition
        .LONG 0
        .LONG 1
```

```
        .SBTTL  SET_CURSOR - SET UP CURSOR REGIONS
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Set two cursor regions for the screen.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_CURSOR:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; Assign one cursor channel using
                   CHAN=CUR_CHAN1       ;  logical name and channel number
        BLBS    R0,10$                  ; Check for success
        BRW     ERROR
10$:    $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; Assign another cursor channel using
                   CHAN=CUR_CHAN2       ;  logical name and channel number
        BLBS    R0,20$
        BRW     ERROR

20$:    MOVL    #IO$C_QV_SETCURSOR,R0   ; Define cursor region code
        $QIOW_S CHAN=CUR_CHAN1,-        ; On cursor channel 1
                FUNC=#IO$_SETMODE,-
                P1 =(R0),-
                P2=#QV$CURSOR1,-        ; Cursor description
                P6=#REGION1             ; Cursor region 1 definition
        BLBS    R0,30$                  ; Check for success
        BRW     ERROR

30$:    MOVL    #IO$C_QV_SETCURSOR,R0   ; Define cursor region code
        $QIOW_S CHAN=CUR_CHAN2,-        ; On cursor channel 2
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#QV$CURSOR2,-        ; Cursor description
                P6=#REGION2             ; Cursor region 2 definition
        BLBS    R0,40$                  ; Check for success
        BRW     ERROR

40$:    RSB

REGION1:                                ; Cursor region 1 definition
        .LONG   20                      ; Lower left corner (ADC)
        .LONG   20
        .LONG   300                     ; Upper right corner (ADC)
        .LONG   300

REGION2:                                ; Cursor region 2 definition
        .LONG   400                     ; Lower left corner (ADC)
        .LONG   400
        .LONG   800                     ; Upper right corner (ADC)
        .LONG   800
```

```
QV$CURSOR1:                             ; 16 * 16 Cursor pattern 1 (solid)
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111

QV$CURSOR2:                             ; 16 * 16 Cursor pattern 2 (cross)
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b1111111111111111
        .WORD   ^b1111111111111111
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000
        .WORD   ^b0000011110000000

        .SBTTL  SET_ONESHOT - SET UP ONE-SHOT QIO
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Set up two 'one-shot' keyboard character reads
;       on keyboard channel 1 by enabling a keyboard region
;       w/o an AST.  The input goes to the typeahead buffer,
;       and the one-shots read it one character at a time.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_ONESHOT:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; Assign a keyboard channel using
                   CHAN=KBD_CHAN1       ;  logical name and channel number
        BLBS    R0,10$                  ; Check for success
        BRW     ERROR
```

D–10 QVSS Programming Example
Programming

```
10$:    MOVL    #IO$C_QV_ENAKB,R0       ; Enable keyboard region code
        $QIOW_S CHAN=KBD_CHAN1,-        ; On keyboard channel 1
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#P2_BLOCK            ; AST specification block
        BLBS    R0,20$                  ; Check for success
        BRW     ERROR

20$:    $QIO_S CHAN=KBD_CHAN1,-         ; Queue 2 one-shot reads
                FUNC=#IO$_READVBLK,-    ; QIO Read code
                IOSB = IOSB_BLOCK,-     ; IOSB block
                ASTADR = ONE_SHOT_AST,- ; AST that reads character
                ASTPRM = #ONESHOT_ACK,- ; Acknowledgment message
                P2 = #IO$C_QV_ENAKB     ; Indicates keyboard list
        BLBS    R0,30$                  ; Check for success
        BRW     ERROR

30$:    $QIO_S CHAN=KBD_CHAN1,-
                FUNC=#IO$_READVBLK,-
                IOSB = IOSB_BLOCK,-
                ASTADR = ONE_SHOT_AST,-
                ASTPRM = #ONESHOT_ACK,-
                P2 = #IO$C_QV_ENAKB
        RSB

P2_BLOCK:                               ; AST specification block 1
        .LONG   0                       ; AST address
        .LONG   0                       ; AST parameter
        .LONG   0                       ; AST delivery mode
        .LONG   0                       ; Input token

ONESHOT_ACK:  ; Acknowledgment message
        .ASCID  /ONE SHOT RECEIVED ON CHANNEL 1/
```

```
        .SBTTL  SET_FRENCH_KB - SET UP A FRENCH KEYBOARD
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Set up a French keyboard on keyboard channel 1.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_FRENCH_KB:
;
; Request that the VC driver use the new keyboard table as the private table
; for keyboard 1.
;
        MOVL    #<IO$C_QV_LOAD_KEY_TABLE>, R0   ; Change the keyboard layout
        $QIOW_S CHAN = KBD_CHAN1, -             ; On keyboard channel 1
                FUNC = #IO$_SETMODE, -
                P1 = (R0), -
                P2 = #KB_LAYOUT_TBL_LEN, -      ; Keyboard table size
                P3 = #KB_LAYOUT_TBL             ; New keyboard table
        BLBS    R0,5$                           ; Check for success
        BRW     ERROR
5$:     RSB

;
; Generate a new keyboard table using macros.  This table will define
; the layout of the characters on the workstation keyboard.
;
        VC$KEYINIT      KB_LAYOUT_TBL   ; Generate the table
;
; Make any changes to the table here.  Because VC$KEYINIT was used to generate
; the table, only the characters that must be changed need to be specified.
; For example, if the "A" key will still be the "A" key in the new layout
; and the various combinations of shift/control/lock are to remain the same, it
; need not be specified again.  (Note that the key definitions do not have to
; be in order.)
;
```

```
VC$KEY    1,<^a/!/>,<^x0B0>,<^x0FF>,<^x0FF>,-
            <^a/!/>,<^x0B0>,<^x0FF>,<^x0FF>
VC$KEY    2,<^a/1/>,<^a/+/>,<^x0FF>,<^x0FF>,-
            <^a/1/>,<^a/+/>,<^X0FF>,<^X0FF>
VC$KEY    3,<^a/2/>,<^a/"/>,<^x000>,<^x000>,-
            <^a/2/>,<^a/"/>,<^x000>,<^x000>
VC$KEY    4,<^a/3/>,<^a/*/>,<^x01B>,<^x01B>,-
            <^a/3/>,<^a/*/>,<^x01B>,<^x01B>
VC$KEY    5,<^a/4/>,<^x0E7>,<^x01C>,<^x01C>,-
            <^a/4/>,<^x0E7>,<^x01C>,<^x01C>
VC$KEY    7,<^a/6/>,<^a/&/>,<^x01E>,<^x01E>,-
            <^a/6/>,<^a/&/>,<^x01E>,<^x01E>
VC$KEY    8,<^a/7/>,<^x02F>,<^x01F>,<^x01F>,-
            <^a/7/>,<^x02F>,<^x01F>,<^x01F>
VC$KEY    9,<^a/8/>,<^a/(/>,<^x07F>,<^x07F>,-
            <^a/8/>,<^a/(/>,<^x07F>,<^x07F>
VC$KEY    10,<^a/9/>,<^a/)/>,<^x0FF>,<^x0FF>,-
            <^a/9/>,<^a/)/>,<^x0FF>,<^x0FF>
VC$KEY    11,<^a/0/>,<^a/=/>,<^x0FF>,<^x0FF>,-
            <^a/0/>,<^a/=/>,<^x0FF>,<^x0FF>
VC$KEY    12,<^x081>,<^a/?/>,<^x0FF>,<^x0FF>,-   ; Diacritical (')
            <^x081>,<^a/?/>,<^x0FF>,<^x0FF>
VC$KEY    13,<^x082>,<^x083>,<^x01E>,<^x01E>,-   ; Diacriticals (` ^)
            <^X082>,<^x083>,<^x0FF>,<^x0FF>
VC$KEY    19,<^a/z/>,<^a/Z/>,<^x01A>,<^x01A>,-
            <^a/z/>,<^a/Z/>,<^x01A>,<^x01A>
VC$KEY    24,<^x0E8>,<^x0FC>,<^x0FF>,<^x0FF>,-
            <^x0E8>,<^x0FC>,<^x0FF>,<^x0FF>
VC$KEY    25,<^x080>,<^x084>,<^x0FF>,<^x0FF>,-   ; Diacriticals (" ~)
            <^x080>,<^x084>,<^x0FF>,<^x0FF>
VC$KEY    35,<^x0E9>,<^x0F6>,<^x0FF>,<^x0FF>,-
            <^x0E9>,<^x0F6>,<^x0FF>,<^x0FF>
VC$KEY    36,<^x0E0>,<^x0E4>,<^x0FF>,<^x0FF>,-
            <^x0E0>,<^x0E4>,<^x0FF>,<^x0FF>
VC$KEY    37,<^a/$/>,<^x0A3>,<^x0FF>,<^x0FF>,-
            <^a/!/>,<^x0B0>,<^x0FF>,<^x0FF>
VC$KEY    46,<^a/,/>,<^a/;/>,<^x0FF>,<^x0FF>,-
            <^a/,/>,<^a/;/>,<^x0FF>,<^x0FF>

VC$KEY    47,<^a/./>,<^a/:/>,<^x0FF>,<^x0FF>,-
            <^a/./>,<^a/:/>,<^x0FF>,<^x0FF>
VC$KEY    48,<^a/-/>,<^a/_/>,<^x0FF>,<^x0FF>,-
            <^a/-/>,<^a/_/>,<^x0FF>,<^x0FF>
VC$KEY    39,<^a/y/>,<^a/Y/>,<^x019>,<^x019>,-
            <^a/y/>,<^a/Y/>,<^x019>,<^x019>

VC$KEYEND       KB_LAYOUT_TBL_LEN       ; End the table,
                                        ;  and determine its length
```

```
        .SBTTL  SET_COMPOSE3_TABLE - SET UP A THREE-STROKE COMPOSE TABLE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Set up a private three-stroke compose table on keyboard channel 1.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_COMPOSE3_TABLE:
;
; Request that the VC driver use the new three-stroke compose table as the
; private table for keyboard 1.
;
        MOVL    #<IO$C_QV_LOAD_COMPOSE_TABLE>, R0 ; Change the compose table
        $QIOW_S CHAN = KBD_CHAN1, -               ; On keyboard channel 1
                FUNC = #IO$_SETMODE, -
                P1 = (R0), -
                P4 = #COMPOSE3_TBL_LEN, -         ; Three-stroke table size
                P5 = #COMPOSE3_TBL                ; New three-stroke table
        BLBS    R0,5$                             ; Check for success
        BRW     ERROR
5$:     RSB

;
; Generate a new three-stroke compose table.  This table will define the new
; compose sequences for a keyboard region (This example shows a subset of
; the standard three-stroke compose sequences - the complete default
; three-stroke table is in Appendix G.)
;
        VC$COMPOSE_KEYINIT      COMPOSE3_TBL    ; Generate an empty table
;
; Fill the table here.
;
        VC$COMPOSE_KEY  <^a/A/>,<^a/"/>,,<^xc4>
        VC$COMPOSE_KEY  <^a/A/>,<^a/'/>,,<^xc1>
        VC$COMPOSE_KEY  <^a/A/>,<^a/*/>,,<^xc5>
        VC$COMPOSE_KEY  <^a/A/>,<^a/A/>,<@>
        VC$COMPOSE_KEY  <^a/A/>,<^a/E/>,,<^xc6>         ; Order sensitive
        VC$COMPOSE_KEY  <^a/A/>,<^a/^/>,,<^xc2>
        VC$COMPOSE_KEY  <^a/A/>,<^a/_/>,,<^xaa>
        VC$COMPOSE_KEY  <^a/A/>,<^a/`/>,,<^xc0>
        VC$COMPOSE_KEY  <^a/A/>,<^a/~/>,,<^xc3>
        VC$COMPOSE_KEY  <^a/A/>,<^x80>,,<^xc4>
        VC$COMPOSE_KEY  <^a/A/>,<^xb0>,,<^xc5>

        VC$COMPOSE_KEY  <^a/C/>,<^a/,/>,,<^xc7>
        VC$COMPOSE_KEY  <^a/C/>,<^a\/\>,,<^xa2>
        VC$COMPOSE_KEY  <^a/C/>,<^a/O/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/C/>,<^a/0/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/C/>,<^a/|/>,,<^xa2>

        VC$COMPOSE_KEYEND  COMPOSE3_TBL_LEN     ; End the table,
                                                ;  and determine its length
```

```
        .SBTTL  SET_COMPOSE2_TABLE - SET UP A TWO-STROKE COMPOSE TABLE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Set up a private two-stroke compose table on keyboard channel 1.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_COMPOSE2_TABLE:
;
; Request that the VC driver use the new two-stroke compose table as the
; private table for keyboard 1.
;
        MOVL    #<IO$C_QV_LOAD_COMPOSE_TABLE>, R0 ; Change the compose table
        $QIOW_S CHAN = KBD_CHAN1, -             ; On keyboard channel 1
                FUNC = #IO$_SETMODE, -
                P1 = (R0), -
                P2 = #COMPOSE2_TBL_LEN, -       ; Two-stroke table size
                P3 = #COMPOSE2_TBL              ; New two-stroke table
        BLBS    R0,5$                           ; Check for success
        BRW     ERROR
5$:     RSB

;
; Generate a new two-stroke compose table.  This table will define the new
; compose sequences for a keyboard region (This example actually shows the
; default table of two-stroke compose sequences).
;
        VC$COMPOSE_KEYINIT      COMPOSE2_TBL,COMPOSE_2=YES
;
; Diaeresis mark
;
        VC$COMPOSE_KEY  <^x80>,<^a/ />,<">
        VC$COMPOSE_KEY  <^x80>,<^a/A/>,,<^xc4>
        VC$COMPOSE_KEY  <^x80>,<^a/E/>,,<^xcb>
        VC$COMPOSE_KEY  <^x80>,<^a/I/>,,<^xcf>
        VC$COMPOSE_KEY  <^x80>,<^a/O/>,,<^xd6>
        VC$COMPOSE_KEY  <^x80>,<^a/U/>,,<^xdc>
        VC$COMPOSE_KEY  <^x80>,<^a/Y/>,,<^xdd>
        VC$COMPOSE_KEY  <^x80>,<^a/a/>,,<^xe4>
        VC$COMPOSE_KEY  <^x80>,<^a/e/>,,<^xeb>
        VC$COMPOSE_KEY  <^x80>,<^a/i/>,,<^xef>
        VC$COMPOSE_KEY  <^x80>,<^a/o/>,,<^F6>
        VC$COMPOSE_KEY  <^x80>,<^a/u/>,,<^xfc>
        VC$COMPOSE_KEY  <^x80>,<^a/y/>,,<^xfd>
```

```
;
; - - -
;
        VC$COMPOSE_KEY  <^x81>,<^a/ />,<'>
        VC$COMPOSE_KEY  <^x81>,<^a/A/>,,<^xc1>
        VC$COMPOSE_KEY  <^x81>,<^a/E/>,,<^xc9>
        VC$COMPOSE_KEY  <^x81>,<^a/I/>,,<^xcd>
        VC$COMPOSE_KEY  <^x81>,<^a/O/>,,<^xd3>
        VC$COMPOSE_KEY  <^x81>,<^a/U/>,,<^xda>
        VC$COMPOSE_KEY  <^x81>,<^a/a/>,,<^xe1>
        VC$COMPOSE_KEY  <^x81>,<^a/e/>,,<^xe9>
        VC$COMPOSE_KEY  <^x81>,<^a/i/>,,<^xed>
        VC$COMPOSE_KEY  <^x81>,<^a/o/>,,<^xf3>
        VC$COMPOSE_KEY  <^x81>,<^a/u/>,,<^xfa>

        VC$COMPOSE_KEY  <^x82>,<^a/ />,<`>
        VC$COMPOSE_KEY  <^x82>,<^a/A/>,,<^xc0>
        VC$COMPOSE_KEY  <^x82>,<^a/E/>,,<^xc8>
        VC$COMPOSE_KEY  <^x82>,<^a/I/>,,<^xcc>
        VC$COMPOSE_KEY  <^x82>,<^a/O/>,,<^xd2>
        VC$COMPOSE_KEY  <^x82>,<^a/U/>,,<^xd9>
        VC$COMPOSE_KEY  <^x82>,<^a/a/>,,<^xe0>
        VC$COMPOSE_KEY  <^x82>,<^a/e/>,,<^xe8>
        VC$COMPOSE_KEY  <^x82>,<^a/i/>,,<^xec>
        VC$COMPOSE_KEY  <^x82>,<^a/o/>,,<^xf2>
        VC$COMPOSE_KEY  <^x82>,<^a/u/>,,<^xf9>

        VC$COMPOSE_KEY  <^x83>,<^a/ />,<^>
        VC$COMPOSE_KEY  <^x83>,<^a/A/>,,<^xc2>
        VC$COMPOSE_KEY  <^x83>,<^a/E/>,,<^xca>
        VC$COMPOSE_KEY  <^x83>,<^a/I/>,,<^xce>
        VC$COMPOSE_KEY  <^x83>,<^a/O/>,,<^xd4>
        VC$COMPOSE_KEY  <^x83>,<^a/U/>,,<^xdb>
        VC$COMPOSE_KEY  <^x83>,<^a/a/>,,<^xe2>
        VC$COMPOSE_KEY  <^x83>,<^a/e/>,,<^xea>
        VC$COMPOSE_KEY  <^x83>,<^a/i/>,,<^xee>
        VC$COMPOSE_KEY  <^x83>,<^a/o/>,,<^xf4>
        VC$COMPOSE_KEY  <^x83>,<^a/u/>,,<^xfb>

        VC$COMPOSE_KEY  <^x84>,<^a/ />,<~>
        VC$COMPOSE_KEY  <^x84>,<^a/A/>,,<^xc3>
        VC$COMPOSE_KEY  <^x84>,<^a/N/>,,<^xd1>
        VC$COMPOSE_KEY  <^x84>,<^a/O/>,,<^xd5>
        VC$COMPOSE_KEY  <^x84>,<^a/a/>,,<^xe3>
        VC$COMPOSE_KEY  <^x84>,<^a/n/>,,<^xf1>
        VC$COMPOSE_KEY  <^x84>,<^a/o/>,,<^xf5>
        VC$COMPOSE_KEY  <^x84>,<^a/u/>,,<^xfc>

        VC$COMPOSE_KEY  <^x85>,<^a/ />,,<^xb0>
        VC$COMPOSE_KEY  <^x85>,<^a/A/>,,<^xc5>
        VC$COMPOSE_KEY  <^x85>,<^a/a/>,,<^xe5>

        VC$COMPOSE_KEYEND  COMPOSE2_TBL_LEN     ; End the table,
                                                ;  and determine its length
```

```
        .SBTTL  SET_KBDAST - SET UP KEYBOARD REGIONS
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Enable several keyboard regions.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_KBDAST:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; Assign keyboard channel 2 using
                   CHAN=KBD_CHAN2       ;  logical name and channel number
        BLBS    R0,10$                  ; Check for success
        BRW     ERROR

10$:    MOVL    #IO$C_QV_MODIFYKB,R0    ; Modify the keyboard
        $QIOW_S CHAN=KBD_CHAN1,-        ; On keyboard channel 1
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#P2_BLOCK1,-         ; AST specification block
                P3=#P3_BLOCK            ; Control AST block
        BLBS    R0,20$                  ; Check for success
        BRW     ERROR

20$:    MOVL    #IO$C_QV_ENAKB,R0       ; Enable a keyboard
        $QIOW_S CHAN=KBD_CHAN2,-        ; On keyboard channel 2
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#P2_BLOCK2,-         ; AST specification block
                P3=#P3_BLOCK            ; Control AST block
        BLBS    R0,30$                  ; Check for success
        BRW     ERROR
30$:    RSB

P2_BLOCK1:                              ; AST specification block 1
        .LONG   KBD_AST                 ; Keyboard AST address
        .LONG   ACK1                    ; AST parameter
        .LONG   0                       ; AST delivery mode
        .LONG   CHARACTER               ; Input token
ACK1:   .ASCID  /INPUT ACKNOWLEDGED CHANNEL 1/ ; Acknowledgment 1

P2_BLOCK2:                              ; AST specification block 2
        .LONG   KBD_AST
        .LONG   ACK2
        .LONG   0
        .LONG   CHARACTER
ACK2:   .ASCID  /INPUT ACKNOWLEDGED CHANNEL 2/

P3_BLOCK:                               ; Control AST specification block
        .LONG   CTL_AST                 ; Control AST address
        .LONG   0                       ; AST parameter
        .LONG   0                       ; AST delivery mode
        .LONG   0                       ; Must be zero
```

```
        .SBTTL  SET_BUTTONAST - ENABLE POINTER BUTTON REGION
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Enable a pointer button region, and specify an AST address.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_BUTTONAST:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-    ; Assign a button channel using
                   CHAN=BUT_CHAN         ;  logical name and channel number
        BLBS    R0,10$                   ; Check for success
        BRW     ERROR

10$:    MOVL    #IO$C_QV_ENABUTTON,R0    ; Enable button transitions
        $QIOW_S CHAN=BUT_CHAN,-          ; On the button channel
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#BUT_BLOCK,-          ; AST specification
                P6=#BUT_REGION           ; Associated button region
        BLBS    R0,20$                   ; Check for success
        BRW     ERROR
20$:    RSB

BUT_BLOCK:                               ; Button AST specification block
        .LONG   BUT_AST                  ; AST address
        .LONG   0                        ; AST parameter
        .LONG   0                        ; Access mode
        .LONG   BUTTON                   ; Button information longword

BUT_REGION:                              ; Button AST region
        .LONG   20                       ; Lower left corner (ADC)
        .LONG   20
        .LONG   300                      ; Upper right corner (ADC)
        .LONG   300
```

```
        .SBTTL  SET_FNKEYAST - ENABLE F5 FUNCTION KEY
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Enable a function key, and specify an AST address.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_FNKEYAST:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; Assign function key channel using
                   CHAN=FNKEY_F5_CHAN   ;  logical name and channel number
        BLBS    R0,10$                  ; Check for success
        BRW     ERROR

10$:    MOVL    #IO$C_QV_ENAFNKEY,R0    ; Enable function key
        $QIOW_S CHAN=FNKEY_F5_CHAN,-    ; On function key channel
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#FNKEY_BLOCK,-       ; AST specification block
                P3=#QV$M_KEY_F5         ; Modifier indicating F5 key
        BLBS    R0,30$                  ; Check for success
        BRW     ERROR

30$:    RSB

FNKEY_BLOCK:                            ; Function key AST specification block
        .LONG   F5_AST                  ; AST address
        .LONG   F5_ACK                  ; AST parameter
        .LONG   0                       ; Access mode
        .LONG   CHARACTER               ; Input token

F5_ACK: .ASCID  /FUNCTION KEY F5 HAS BEEN PRESSED/ ; Acknowledgment message

        .SBTTL  SET_MOUSEAST - ENABLE POINTER MOVEMENT REGION
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Enable pointer movement ASTs for a region.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SET_MOUSEAST:
        $ASSIGN_S  DEVNAM=WS_DEVNAM,-   ; Assign pointer channel using
                   CHAN=MOUSE_CHAN      ;  logical name and channel number
        BLBS    R0,10$                  ; Check for success
        BRW     ERROR
```

```
10$:    MOVL    #IO$C_QV_MOUSEMOV,R0    ; Enable pointer movement
        $QIOW_S CHAN=MOUSE_CHAN,-       ; On pointer channel
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#MOUSE_BLOCK,-       ; AST specification block
                P6=#MOUSE_REGION        ; Associated region
        BLBS    R0,20$                  ; Check for success
        BRW     ERROR
20$:    RSB

MOUSE_BLOCK:                            ; Pointer region AST specification block
        .LONG   MOUSE_AST               ; AST address
        .LONG   MOUSE_ACK               ; AST parameter
        .LONG   0                       ; Access mode
        .LONG   MOUSE_XY                ; New cursor position

MOUSE_REGION:                           ; Pointer region
        .LONG   400                     ; Lower left corner (ADC)
        .LONG   400
        .LONG   800                     ; Upper right corner (ADC)
        .LONG   800

MOUSE_ACK:
        .ASCID  /POINTER MOVEMENT DETECTED/ ; Acknowledgment message

        .SBTTL  SIMULATE_INPUT - SIMULATE INPUT ON KEYBOARD 2
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Simulate keyboard input on keyboard channel 2.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
SIMULATE_INPUT:
        MOVL    #IO$C_QV_SIMULATE,R0    ; Simulate keyboard input
        $QIOW_S CHAN=KBD_CHAN2,-        ; On keyboard channel 2
                FUNC=#IO$_SETMODE,-
                P1=(R0),-
                P2=#SIM_ACK,-           ; Acknowledgment
                P3=#0                   ; No pointer repositioning
        BLBS    R0,20$                  ; Check for success
        BRW     ERROR
20$:    RSB

SIM_ACK:
        .ASCID  /This input SIMULATED on chan 2./  ; Acknowledgment message


;
; The following code contains all the ASTs specified in the above QIOs
:
```

```
        .SBTTL  ONE_SHOT_AST - ONE_SHOT AST ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       This is the AST routine specified by the 'one-shot' read QIO.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
ONE_SHOT_AST:
        .WORD
        PUSHL   4(AP)                   ; Send acknowledgment message
        CALLS   #1,G^LIB$PUT_LINE
        BLBS    R0, 10$
        BRW     ERROR

10$:    PUSHAL  DESC1                   ; Send character typed,
        CALLS   #1,G^LIB$PUT_LINE       ;  descriptor points to IOSB

        RET

        .SBTTL  MOUSE_AST - POINTER AST ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       This is the AST routine specified by the enable pointer movement QIO.
;
; INPUT PARAMETERS:
;       POINTER_XY - Contains the current X and Y coordinates for the pointer.
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
MOUSE_AST:
        .WORD
        PUSHL   4(AP)                   ; Send acknowledgment message
        CALLS   #1,G^LIB$PUT_LINE
        BLBS    R0, 10$
5$:     BRW     ERROR

10$:    RET
```

```
        .SBTTL  KBD_AST - KEYBOARD AST ROUTINE
        .SBTTL  F5_AST - FUNCTION KEY F5 AST ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       This is the AST routine specified by the enable keyboard QIO and
;       by the enable function key F5 QIO.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
KBD_AST:
F5_AST:
        .WORD
        PUSHL   4(AP)                   ; Send acknowledgment message
        CALLS   #1,G^LIB$PUT_LINE
        BLBS    R0, 10$
5$:     BRW     ERROR

10$:    CMPW    #KEY$C_F5,CHARACTER     ; Was F5 typed?
        BNEQ    20$
        BSBW    CYCLE_KBD               ; Cycle the keyboard list
        BRB     40$                     ;  and exit

20$:    PUSHAL  DESC                    ; Send character typed
        CALLS   #1,G^LIB$PUT_LINE
        BLBC    R0,5$

        CMPB    #^A/C/,CHARACTER        ; Was a "C" typed?
        BNEQ    30$
        BSBW    CYCLE_KBD               ; Cycle the keyboard list
        BRB     40$

30$:    CMPB    #^A/F/,CHARACTER        ; Was an "F" typed?
        BNEQ    40$
        $SETEF_S EFN=#2                 ; Yes, exit program

40$:    RET

        .SBTTL  CTL_AST - CONTROL AST ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       This is the control AST routine.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
```

```
CTL_AST:
        .WORD
        PUSHAL  CYCLE_ACK               ; Send acknowledgment message
        CALLS   #1,G^LIB$PUT_LINE
        BLBS    R0, 10$
5$:     BRW     ERROR
10$:    RET

CYCLE_ACK:
        .ASCID  /KEYBOARD HAS BEEN CYCLED/ ; Acknowledgment message

        .SBTTL  BUT_AST - BUTTON AST ROUTINE
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       This is the AST routine specified by the enable pointer movement QIO.
;
; INPUT PARAMETERS:
;       BUTTON - Status of the pointer buttons.
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
BUT_AST:
        .WORD
        CMPW    #KEY$C_BUTTON_1,BUTTON  ; Was BUTTON 1 changed?
        BNEQ    10$                     ;  (predefined constant)
        PUSHAL  BUT1_ACK
        BRB     50$

10$:    CMPW    #KEY$C_BUTTON_2,BUTTON  ; Was BUTTON 2 changed?
        BNEQ    20$
        PUSHAL  BUT2_ACK
        BRB     50$

20$:    CMPW    #KEY$C_BUTTON_3,BUTTON  ; Was BUTTON 3 changed?
        BNEQ    100$
        PUSHAL  BUT3_ACK

50$:    CALLS   #1,G^LIB$PUT_LINE       ; Send correct acknowledgment
        BLBS    R0,60$
        BRW     ERROR

60$:    PUSHAL  BUTUP_ACK               ; Assume button was released
        BBC     #31,BUTTON,70$          ; If clear then button released
        PUSHAL  BUTDOWN_ACK             ; Otherwise, button was depressed
70$:    CALLS   #1,G^LIB$PUT_LINE       ; Send acknowledgment message
        BLBS    R0,100$
        BRW     ERROR
100$:   RET
```

```
;               Acknowledgment messages
BUT1_ACK:
        .ASCID  /POINTER BUTTON 1 TRANSITION HAS BEEN DETECTED/
BUT2_ACK:
        .ASCID  /POINTER BUTTON 2 TRANSITION HAS BEEN DETECTED/
BUT3_ACK:
        .ASCID  /POINTER BUTTON 3 TRANSITION HAS BEEN DETECTED/
BUTUP_ACK:
        .ASCID  /BUTTON IS UP/
BUTDOWN_ACK:
        .ASCID  /BUTTON IS DOWN/

        .SBTTL  CYCLE_KBD - CYCLE KEYBOARD REGION
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       This routine cycles the keyboard to the next keyboard region.
;
; INPUT PARAMETERS:
;       NONE
;
; OUTPUT PARAMETERS:
;       NONE
;
;--
;
CYCLE_KBD:
        MOVL    #<IO$C_QV_ENAKB!IO$M_QV_CYCLE>,R0 ; Use keyboard cycle modifier
        $QIOW_S CHAN=KBD_CHAN1,-                   ; On keyboard channel 1
                FUNC=#IO$_SETMODE,-
                P1= (R0)

        BLBS    R0,40$                  ; Check for success
        BRW     ERROR

40$:    RSB

        .END     START
```

Appendix E

Keyboard Table Macros

```
        .SBTTL  VC$KEYINIT - Macro to generate keyboard table
;++
; VC$KEYINIT  - Initializes the keyboard table
; VC$KEY      - Generates a key table entry for the given position
;
; This macro defines a given key entry for the keyboard position.
; This entry contains the ASCII translation for the lowercase character,
; uppercase character, control (CTRL) for this position, and shift control
; for this position.  In addition it will generate a lock state for this
; position including the character to be used when lock is detected, when
; shift lock is detected, when control lock is detected, and when control
; shift lock is detected.
;--
;
        .Macro  VC$KEY  POSITION,LOWER,SHIFT,CTRL,SHIFT_CTRL,LOCK,-
                        SHIFT_LOCK,CTRL_LOCK,SHIFT_CTRL_LOCK
        .=BASE+<<POSITION-1>*8>

        .BYTE   LOWER                   ; Lowercase character

        .IF BLANK,<SHIFT>               ; Shift (uppercase) character
        .BYTE   LOWER&<^C<1@5>>
        .IFF
        .BYTE   SHIFT
        .ENDC

        .IF BLANK,<CTRL>                ; Control key and character key
        .BYTE   LOWER&<^C<3@5>>
        .IFF
        .BYTE   CTRL
        .ENDC
```

```
.IF  BLANK,<SHIFT_CTRL>          ; Shift, control and character key
.BYTE   LOWER&<^C<3@5>>          ; Usually same as control
.IFF
.BYTE   SHIFT_CTRL
.ENDC

.IF BLANK,<LOCK>                 ; Lock and character keys
.BYTE   LOWER&<^C<1@5>>          ; Usually the same as shift
.IFF
.BYTE   LOCK
.ENDC

.IF BLANK,<SHIFT_LOCK>           ; Shift, lock and character keys
.BYTE   LOWER&<^C<1@5>>          ; Usually the same as shift
.IFF
.BYTE   SHIFT_LOCK
.ENDC

.IF BLANK,<CTRL_LOCK>            ; Control, lock and character keys
.BYTE   LOWER&<^C<3@5>>          ; Usually the same as control
.IFF
.BYTE   CTRL_LOCK
.ENDC

.IF  BLANK,<SHIFT_CTRL_LOCK>     ; Shift, control, lock & char keys
.BYTE   LOWER&<^C<3@5>>          ; Usually the same as control
.IFF
.BYTE   SHIFT_CTRL_LOCK
.ENDC

.ENDM   VC$KEY
```

```
        .Macro  VC$KEYINIT  NAME,VERSION=1,INTERNAL
        .SAVE
        .PSECT  $$$117_'NAME,LONG
        .IF NB  INTERNAL
        .long   2           ; * Used for building internal driver table *
        .ENDC
MAXKEY=49
TBL_START=.
NAME==.
        .QUAD   VERSION
;
; Allocate space for the table. Pre-initialize it to known values (-1).
;
BASE=.
        .REPEAT MAXKEY-1
        .LONG   -1,-1
        .ENDR
ENDBASE=.
;
; Initialize the table to the North American keyboard.
;
        VC$KEY  1,<^a/`/>,<^a/~/>,<^x1e>,<^x1e>,-
                  <^a/`/>,<^a/~/>,<^x1e>,<^x1e>
        VC$KEY  2,<^a/1/>,<^a/!/>,<^x0FF>,<^x0FF>,-
                  <^a/1/>,<^a/!/>,<^x0FF>,<^x0FF>
        VC$KEY  3,<^a/2/>,<^a/@/>,<^x00>,<^x00>,-
                  <^a/2/>,<^a/@/>,<^x00>,<^x00>
        VC$KEY  4,<^a/3/>,<^a/#/>,<^x1b>,<^x1b>,-
                  <^a/3/>,<^a/#/>,<^x1b>,<^x1b>
        VC$KEY  5,<^a/4/>,<^a/$/>,<^x1c>,<^x1c>,-
                  <^a/4/>,<^a/$/>,<^x1c>,<^x1c>
        VC$KEY  6,<^a/5/>,<^a/%/>,<^x1d>,<^x1d>,-
                  <^a/5/>,<^a/%/>,<^x1d>,<^x1d>
        VC$KEY  7,<^a/6/>,<^a/^/>,<^x1e>,<^x1e>,-
                  <^a/6/>,<^a/^/>,<^x1e>,<^x1e>
        VC$KEY  8,<^a/7/>,<^a/&/>,<^x1f>,<^x1f>,-
                  <^a/7/>,<^a/&/>,<^x1f>,<^x1f>
```

```
VC$KEY  9,<^a/8/>,<^a/*/>,<^x7f>,<^x7f>,-
          <^a/8/>,<^a/*/>,<^x7f>,<^x7f>
VC$KEY  10,<^a/9/>,<^a/(/>,<^x0FF>,<^x0FF>,-
           <^a/9/>,<^a/(/>,<^x0FF>,<^x0FF>
VC$KEY  11,<^a/0/>,<^a/)/>,<^x0FF>,<^x0FF>,-
           <^a/0/>,<^a/)/>,<^x0FF>,<^x0FF>
VC$KEY  12,<^a/-/>,<^a/_/>,<^x1f>,<^x1f>,-
           <^a/-/>,<^a/_/>,<^x1f>,<^x1f>
VC$KEY  13,<^a/=/>,<^a/+/>,<^x0FF>,<^x0FF>,-
           <^a/=/>,<^a/+/>,<^x0FF>,<^x0FF>
VC$KEY  14,<^a/q/>,<^a/Q/>,<^x11>,<^x11>,-
                  ,       ,<^x11>,<^x11>
VC$KEY  15,<^A/w/>,<^a/W/>,<^x17>,<^x17>,-
                  ,       ,<^x17>,<^x17>
VC$KEY  16,<^A/e/>,<^a/E/>,<^x05>,<^x05>,-
                  ,       ,<^x05>,<^x05>
VC$KEY  17,<^A/r/>,<^a/R/>,<^x12>,<^x12>,-
                  ,       ,<^x12>,<^x12>
VC$KEY  18,<^A/t/>,<^a/T/>,<^x14>,<^x14>,-
                  ,       ,<^x14>,<^x14>
VC$KEY  19,<^A/y/>,<^a/Y/>,<^x19>,<^x19>,-
                  ,       ,<^x19>,<^x19>
VC$KEY  20,<^A/u/>,<^a/U/>,<^x15>,<^x15>,-
                  ,       ,<^x15>,<^x15>
VC$KEY  21,<^A/i/>,<^a/I/>,<^x09>,<^x09>,-
                  ,       ,<^x09>,<^x09>
VC$KEY  22,<^A/o/>,<^a/O/>,<^x0F>,<^x0F>,-
                  ,       ,<^x0F>,<^x0F>
VC$KEY  23,<^A/p/>,<^a/P/>,<^x10>,<^x10>,-
                  ,       ,<^x10>,<^x10>
VC$KEY  24,<^A/[/>,<^a/{/>,<^x1b>,<^x1b>,-
           <^a/[/>,<^a/{/>,<^x1b>,<^x1b>
VC$KEY  25,<^A/]/>,<^a/}/>,<^x1d>,<^x1d>,-
           <^a/]/>,<^a/}/>,<^x1d>,<^x1d>
VC$KEY  26,<^A/a/>,<^a/A/>,<^x01>,<^x01>,-
                  ,       ,<^x01>,<^x01>
VC$KEY  27,<^A/s/>,<^a/S/>,<^x13>,<^x13>,-
                  ,       ,<^x13>,<^x13>
```

```
        VC$KEY  28,<^A/d/>,<^a/D/>,<^x04>,<^x04>,-
                        ,       ,<^x04>,<^x04>
        VC$KEY  29,<^A/f/>,<^a/F/>,<^x06>,<^x06>,-
                        ,       ,<^x06>,<^x06>
        VC$KEY  30,<^A/g/>,<^a/G/>,<^x07>,<^x07>,-
                        ,       ,<^x07>,<^x07>
        VC$KEY  31,<^A/h/>,<^a/H/>,<^x08>,<^x08>,-
                        ,       ,<^x08>,<^x08>
        VC$KEY  32,<^A/j/>,<^a/J/>,<^x0A>,<^x0A>,-
                        ,       ,<^x0A>,<^x0A>
        VC$KEY  33,<^A/k/>,<^a/K/>,<^x0B>,<^x0B>,-
                        ,       ,<^x0B>,<^x0B>
        VC$KEY  34,<^A/l/>,<^a/L/>,<^x0C>,<^x0C>,-
                        ,       ,<^x0C>,<^x0C>
        VC$KEY  35,<^x03B>,<^a/:/>,<^x0FF>,<^x0FF>,-
                   <^x03B>,<^a/:/>,<^x0FF>,<^x0FF>
        VC$KEY  36,<^a/'/>,<^a/"/>,<^x0FF>,<^x0FF>,-
                   <^a/'/>,<^a/"/>,<^x0FF>,<^x0FF>
        VC$KEY  37,<^a/\/>,<^a/|/>,<^x1c>,<^x1c>,-
                   <^a/\/>,<^a/|/>,<^x1c>,<^x1c>
        VC$KEY  38,<^x3c>,<^x3e>,<^x0FF>,<^x0FF>,-
                   <^x3c>,<^x3e>,<^x0FF>,<^x0FF>
        VC$KEY  39,<^A/z/>,<^a/Z/>,<^x1A>,<^x1A>,-
                        ,       ,<^x1A>,<^x1A>
        VC$KEY  40,<^A/x/>,<^a/X/>,<^x18>,<^x18>,-
                        ,       ,<^x18>,<^x18>
        VC$KEY  41,<^A/c/>,<^a/C/>,<^x03>,<^x03>,-
                        ,       ,<^x03>,<^x03>
        VC$KEY  42,<^A/v/>,<^a/V/>,<^x16>,<^x16>,-
                        ,       ,<^x16>,<^x16>
        VC$KEY  43,<^A/b/>,<^a/B/>,<^x02>,<^x02>,-
                        ,       ,<^x02>,<^x02>
        VC$KEY  44,<^A/n/>,<^a/N/>,<^x0E>,<^x0E>,-
                        ,       ,<^x0E>,<^x0E>
        VC$KEY  45,<^A/m/>,<^a/M/>,<^x0D>,<^x0D>,-
                        ,       ,<^x0D>,<^x0D>
        VC$KEY  46,<^a/,/>,<^a/,/>,<^x0FF>,<^x0FF>,-
                   <^a/,/>,<^a/,/>,<^x0FF>,<^x0FF>
        VC$KEY  47,<^a/./>,<^a/./>,<^x0FF>,<^x0FF>,-
                   <^a/./>,<^a/./>,<^x0FF>,<^x0FF>
        VC$KEY  48,<^x2f>,<^a/?/>,<^x1f>,<^x1f>,-
                   <^x2f>,<^a/?/>,<^x1f>,<^x1f>
        .endm   VC$KEYINIT

        .MACRO  VC$KEYEND TABLE_SIZE
=ENDBASE                                ; PC points to first byte after table
```

```
.IF NOT_BLANK TABLE_SIZE
TABLE_SIZE == ENDBASE - TBL_START       ;Size of table
.ENDC
.RESTORE
.ENDM   VC$KEYEND
```

Appendix F

Compose Table Macros

This appendix contains the macros used to construct a compose table.

```
        .SBTTL  VC$COMPOSE_KEYINI - Macro to initialize a compose sequence table
;++
; MACRO VC$COMPOSE_KEYINI  - Macro to initialize a compose sequence table
;
; INPUTS
;       NAME = TABLE NAME TO GENERATE
;       COMPOSE_2 = YES - IF THIS IS A 2 CHARACTER COMPOSE TABLE
;                   BLANK - IF A 3 CHARACTER COMPOSE TABLE
;       VERSION = TABLE VERSION
;       INTERNAL = YES - IF THIS MACRO IS BEING CALLED TO BUILD INTERNAL TABLE.
;                  BLANK - IF THIS MACRO IS BEING CALLED NORMALLY.
;--
;
        .Macro  VC$COMPOSE_KEYINIT  NAME,COMPOSE_2,VERSION=1,INTERNAL
        .SAVE
        .PSECT  $$$118_'NAME'_A
        .IF NB  INTERNAL
        .long   2               ; *** Used by driver, NOT for regular table ***
        .ENDC

NAME==.
        .save                   ; Save attributes
        .psect $$$118_'NAME'_B  ; Go to other psect
        ext_'name=.             ; Save base address
        .restore                ; Restore attributes
```

F–2 Compose Table Macros

```
        .LONG   VERSION
        .IF NB  COMPOSE_2
        i=0
DIA_TAB_'NAME=.
        .repeat <256/32>        ; Get the diacritical table
        .long   0
        dia_init  \i
        i=i+1
        .endr
        .ENDC

        .SBTTL  VC$COMPOSE_KEY - Macro to generate a compose table entry
;++
; MACRO VC$COMPOSE_KEY - Macro to generate a compose table entry
;
; INPUTS
;       INPUT_CHAR1 = FIRST CHARACTER OF COMPOSE SEQUENCE
;       INPUT_CHAR2 = SECOND CHARACTER OF COMPOSE SEQUENCE
;       OUTPUT_SIZE = SIZE OF THE OUTPUT STRING (optional,
;                       if omitted, size will be calculated)
;       OUTPUT_CHAR = OUTPUT STRING
;--
;
        .Macro  VC$COMPOSE_KEY  input_char1,input_char2,output_size,output_char

        .BYTE   INPUT_CHAR1
        .BYTE   INPUT_CHAR2
        .SAVE
        .PSECT  $$$118_'NAME'_B
STR=.
        .IF BLANK <OUTPUT_SIZE>
        .BYTE   1
        .BYTE   OUTPUT_CHAR

        .IFF
        .ASCIC  |OUTPUT_SIZE|
        .ENDC

        .RESTORE
        .WORD   STR-NAME
        .IF NB COMPOSE_2
        I=INPUT_CHAR1/32
        J=INPUT_CHAR1-<I*32>
        DIA     \I,\J
        .ENDC
        .endm   VC$COMPOSE_KEY
```

```
;++
; MACRO VC$COMPOSE_KEYEND - Macro to end a compose table
;
; INPUTS
;       TABLE_SIZE - (optional) Location to store size of table
;--
;
        .MACRO  VC$COMPOSE_KEYEND  TABLE_SIZE
        .LONG   -1
        .IIF NB,COMPOSE_2, DIA_END DIA_TAB_'NAME
        .IF NB  TABLE_SIZE                          ; If table size requested
        TABLE_SIZE == .-name                        ; Get size of first psect
        .psect $$$118_'NAME'_B                      ; Jump to other psect
        TABLE_SIZE == TABLE_SIZE+<.-ext_'name>      ; Add in size of this psect
        .ENDC
        .RESTORE
        .ENDM   VC$COMPOSE_KEYEND

        .ENDM   VC$COMPOSE_KEYINIT

        .SBTTL  MACRO DIA_INIT - Macro to generate diacritical table
;++
; MACRO DIA_INIT - Macro to generate diacritical table
;
; INPUTS
;       N - OFFSET INTO TABLE
;--
;
        .MACRO  DIA_INIT N
        DIA_'N = 0
        .MACRO  DIA     OFFSET,BIT_POS
        .IIF GT OFFSET-N,.ERROR ; OFFSETS CANNOT BE GREATER THAN DIA_MAX
        DIA_'OFFSET=1@BIT_POS!DIA_'OFFSET
        .ENDM   DIA
        .MACRO  DIA_GEN X
        .LONG   DIA_'X
        .ENDM   DIA_GEN
        .MACRO  DIA_END LABEL
        .SAVE
        .=LABEL
        X=0
        .REPEAT N+1
        DIA_GEN \X
        X=X+1
        .ENDR
        .RESTORE
        .ENDM   DIA_END
        .ENDM   DIA_INIT
```

Appendix G

Three-Stroke Compose Table Default Values

This appendix contains the macro that is executed to load the default three-stroke compose table values. (The default two-stroke table is shown in the example in Appendix D.)

```
        VC$COMPOSE_KEYINIT      QV$COMPOSE3_TABLE

;
; sp ! " # $ % & ' ( ) * + , - . /
;
        VC$COMPOSE_KEY  <^a/ />,<^a/"/>,<">
        VC$COMPOSE_KEY  <^a/ />,<^a/'/>,<'>
        VC$COMPOSE_KEY  <^a/ />,<^a/*/>,,<^xb0>
        VC$COMPOSE_KEY  <^a/ />,<^a/^/>,<^>
        VC$COMPOSE_KEY  <^a/ />,<^a/~/>,<~>
        VC$COMPOSE_KEY  <^a/ />,<^xb0>,,<^xb0>

        VC$COMPOSE_KEY  <^a/!/>,<^a/!/>,,<^xa1>
        VC$COMPOSE_KEY  <^a/!/>,<^a/P/>,,<^xb6>
        VC$COMPOSE_KEY  <^a/!/>,<^a/S/>,,<^xa7>
        VC$COMPOSE_KEY  <^a/!/>,<^a/p/>,,<^xb6>
        VC$COMPOSE_KEY  <^a/!/>,<^a/s/>,,<^xa7>

        VC$COMPOSE_KEY  <^a/"/>,<^a/ />,<">
        VC$COMPOSE_KEY  <^a/"/>,<^a/A/>,,<^xc4>
        VC$COMPOSE_KEY  <^a/"/>,<^a/E/>,,<^xcb>
        VC$COMPOSE_KEY  <^a/"/>,<^a/I/>,,<^xcf>
        VC$COMPOSE_KEY  <^a/"/>,<^a/O/>,,<^xd6>
        VC$COMPOSE_KEY  <^a/"/>,<^a/U/>,,<^xdc>
        VC$COMPOSE_KEY  <^a/"/>,<^a/Y/>,,<^xdd>
        VC$COMPOSE_KEY  <^a/"/>,<^a/a/>,,<^xe4>
        VC$COMPOSE_KEY  <^a/"/>,<^a/e/>,,<^xeb>
        VC$COMPOSE_KEY  <^a/"/>,<^a/i/>,,<^xef>
        VC$COMPOSE_KEY  <^a/"/>,<^a/o/>,,<^xf6>
        VC$COMPOSE_KEY  <^a/"/>,<^a/u/>,,<^xfc>
        VC$COMPOSE_KEY  <^a/"/>,<^a/y/>,,<^xfd>
```

```
VC$COMPOSE_KEY  <^a/'/>,<^a/ />,<'>
VC$COMPOSE_KEY  <^a/'/>,<^a/A/>,,<^xc1>
VC$COMPOSE_KEY  <^a/'/>,<^a/E/>,,<^xc9>
VC$COMPOSE_KEY  <^a/'/>,<^a/I/>,,<^xcd>
VC$COMPOSE_KEY  <^a/'/>,<^a/O/>,,<^xd3>
VC$COMPOSE_KEY  <^a/'/>,<^a/U/>,,<^xda>
VC$COMPOSE_KEY  <^a/'/>,<^a/a/>,,<^xe1>
VC$COMPOSE_KEY  <^a/'/>,<^a/e/>,,<^xe9>
VC$COMPOSE_KEY  <^a/'/>,<^a/i/>,,<^xed>
VC$COMPOSE_KEY  <^a/'/>,<^a/o/>,,<^xf3>
VC$COMPOSE_KEY  <^a/'/>,<^a/u/>,,<^xfa>

VC$COMPOSE_KEY  <^a/(/>,<^a/(/>,<[>
VC$COMPOSE_KEY  <^a/(/>,<^a/-/>,<{>

VC$COMPOSE_KEY  <^a/)/>,<^a/)/>,<]>
VC$COMPOSE_KEY  <^a/)/>,<^a/-/>,<}>

VC$COMPOSE_KEY  <^a/*/>,<^a/ />,,<^xb0>
VC$COMPOSE_KEY  <^a/*/>,<^a/A/>,,<^xc5>
VC$COMPOSE_KEY  <^a/*/>,<^a/a/>,,<^xe5>

VC$COMPOSE_KEY  <^a/+/>,<^a/+/>,<#>
VC$COMPOSE_KEY  <^a/+/>,<^a/-/>,,<^xb1>

VC$COMPOSE_KEY  <^a/,/>,<^a/C/>,,<^xc7>
VC$COMPOSE_KEY  <^a/,/>,<^a/c/>,,<^xe7>

VC$COMPOSE_KEY  <^a/-/>,<^a/(/>,<{>
VC$COMPOSE_KEY  <^a/-/>,<^a/)/>,<}>
VC$COMPOSE_KEY  <^a/-/>,<^a/+/>,,<^xb1>
VC$COMPOSE_KEY  <^a/-/>,<^a/L/>,,<^xa3>
VC$COMPOSE_KEY  <^a/-/>,<^a/Y/>,,<^xa5>
VC$COMPOSE_KEY  <^a/-/>,<^a/l/>,,<^xa3>
VC$COMPOSE_KEY  <^a/-/>,<^a/y/>,,<^xa5>

VC$COMPOSE_KEY  <^a/./>,<^a/^/>,,<^xb7>

VC$COMPOSE_KEY  <^a\/\>,<^a\/\>,<\>
VC$COMPOSE_KEY  <^a\/\>,<^x3c>,<\>
VC$COMPOSE_KEY  <^a\/\>,<^a/C/>,,<^xa2>
VC$COMPOSE_KEY  <^a\/\>,<^a/O/>,,<^xd8>
VC$COMPOSE_KEY  <^a\/\>,<^a/U/>,,<^xb5>          ; order sensitive
VC$COMPOSE_KEY  <^a\/\>,<^a/^/>,,<^a/|/>
VC$COMPOSE_KEY  <^a\/\>,<^a/c/>,,<^xa2>
VC$COMPOSE_KEY  <^a\/\>,<^a/o/>,,<^xf8>
VC$COMPOSE_KEY  <^a\/\>,<^a/u/>,,<^xb5>          ; order sensitive
```

```
;
; 0 1 2 3 4 5 6 7 8 9  : ; < = > ?
;
        VC$COMPOSE_KEY  <^a/0/>,<^a/C/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/0/>,<^a/S/>,,<^xa7>
        VC$COMPOSE_KEY  <^a/0/>,<^a/X/>,,<^xa8>
        VC$COMPOSE_KEY  <^a/0/>,<^a/^/>,,<^xb0>
        VC$COMPOSE_KEY  <^a/0/>,<^a/c/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/0/>,<^a/s/>,,<^xa7>
        VC$COMPOSE_KEY  <^a/0/>,<^a/x/>,,<^xa8>

        VC$COMPOSE_KEY  <^a/1/>,<^a/2/>,,<^xbd>          ; order sensitive
        VC$COMPOSE_KEY  <^a/1/>,<^a/4/>,,<^xbc>          ; order sensitive
        VC$COMPOSE_KEY  <^a/1/>,<^a/^/>,,<^xb9>

        VC$COMPOSE_KEY  <^a/2/>,<^a/^/>,,<^xb2>

        VC$COMPOSE_KEY  <^a/3/>,<^a/^/>,,<^xb3>

        VC$COMPOSE_KEY  <^x3c>,<^a\/\>,<\>
        VC$COMPOSE_KEY  <^x3c>,<^x3c>,,<^xab>

        VC$COMPOSE_KEY  <^a/=/>,<^a/L/>,,<^xa3>
        VC$COMPOSE_KEY  <^a/=/>,<^a/Y/>,,<^xa5>
        VC$COMPOSE_KEY  <^a/=/>,<^a/l/>,,<^xa3>
        VC$COMPOSE_KEY  <^a/=/>,<^a/y/>,,<^xa5>

        VC$COMPOSE_KEY  <^x3e>,<^x3e>,,<^xbb>

        VC$COMPOSE_KEY  <^a/?/>,<^a/?/>,,<^xbf>

;
; A to Z
;
        VC$COMPOSE_KEY  <^a/A/>,<^a/"/>,,<^xc4>
        VC$COMPOSE_KEY  <^a/A/>,<^a/'/>,,<^xc1>
        VC$COMPOSE_KEY  <^a/A/>,<^a/*/>,,<^xc5>
        VC$COMPOSE_KEY  <^a/A/>,<^a/A/>,<@>
        VC$COMPOSE_KEY  <^a/A/>,<^a/E/>,,<^xc6>          ; order sensitive
        VC$COMPOSE_KEY  <^a/A/>,<^a/^/>,,<^xc2>
        VC$COMPOSE_KEY  <^a/A/>,<^a/_/>,,<^xaa>
        VC$COMPOSE_KEY  <^a/A/>,<^a/`/>,,<^xc0>
        VC$COMPOSE_KEY  <^a/A/>,<^a/~/>,,<^xc3>
        VC$COMPOSE_KEY  <^a/A/>,<^x80>,,<^xc4>
        VC$COMPOSE_KEY  <^a/A/>,<^xb0>,,<^xc5>

        VC$COMPOSE_KEY  <^a/C/>,<^a/,/>,,<^xc7>
        VC$COMPOSE_KEY  <^a/C/>,<^a\/\>,,<^xa2>
        VC$COMPOSE_KEY  <^a/C/>,<^a/O/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/C/>,<^a/O/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/C/>,<^a/|/>,,<^xa2>
```

```
VC$COMPOSE_KEY  <^a/E/>,<^a/"/>,,<^xcb>
VC$COMPOSE_KEY  <^a/E/>,<^a/'/>,,<^xc9>
VC$COMPOSE_KEY  <^a/E/>,<^a/^/>,,<^xca>
VC$COMPOSE_KEY  <^a/E/>,<^a/`/>,,<^xc8>
VC$COMPOSE_KEY  <^a/E/>,<^x80>,,<^xcb>

VC$COMPOSE_KEY  <^a/I/>,<^a/"/>,,<^xcf>
VC$COMPOSE_KEY  <^a/I/>,<^a/'/>,,<^xcd>
VC$COMPOSE_KEY  <^a/I/>,<^a/^/>,,<^xce>
VC$COMPOSE_KEY  <^a/I/>,<^a/`/>,,<^xcc>
VC$COMPOSE_KEY  <^a/I/>,<^x80>,,<^xcf>

VC$COMPOSE_KEY  <^a/L/>,<^a/-/>,,<^xa3>
VC$COMPOSE_KEY  <^a/L/>,<^a/=/>,,<^xa3>

VC$COMPOSE_KEY  <^a/N/>,<^a/~/>,,<^xd1>

VC$COMPOSE_KEY  <^a/O/>,<^a/"/>,,<^xd6>
VC$COMPOSE_KEY  <^a/O/>,<^a/'/>,,<^xd3>
VC$COMPOSE_KEY  <^a/O/>,<^a\/\>,,<^xd8>
VC$COMPOSE_KEY  <^a/O/>,<^a/C/>,,<^xa9>
VC$COMPOSE_KEY  <^a/O/>,<^a/E/>,,<^xd7>          ;  order sensitive
VC$COMPOSE_KEY  <^a/O/>,<^a/S/>,,<^xa7>
VC$COMPOSE_KEY  <^a/O/>,<^a/X/>,,<^xa8>
VC$COMPOSE_KEY  <^a/O/>,<^a/^/>,,<^xd4>
VC$COMPOSE_KEY  <^a/O/>,<^a/_/>,,<^xba>
VC$COMPOSE_KEY  <^a/O/>,<^a/`/>,,<^xd2>
VC$COMPOSE_KEY  <^a/O/>,<^a/~/>,,<^xd5>
VC$COMPOSE_KEY  <^a/O/>,<^x80>,,<^xd6>

VC$COMPOSE_KEY  <^a/P/>,<^a/!/>,,<^xb6>

VC$COMPOSE_KEY  <^a/S/>,<^a/!/>,,<^xa7>
VC$COMPOSE_KEY  <^a/S/>,<^a/O/>,,<^xa7>
VC$COMPOSE_KEY  <^a/S/>,<^a/O/>,,<^xa7>

VC$COMPOSE_KEY  <^a/U/>,<^a/"/>,,<^xdc>
VC$COMPOSE_KEY  <^a/U/>,<^a/'/>,,<^xda>
VC$COMPOSE_KEY  <^a/U/>,<^a/^/>,,<^xdb>
VC$COMPOSE_KEY  <^a/U/>,<^a/`/>,,<^xd9>
VC$COMPOSE_KEY  <^a/U/>,<^x80>,,<^xdc>

VC$COMPOSE_KEY  <^a/X/>,<^a/O/>,,<^xa8>
VC$COMPOSE_KEY  <^a/X/>,<^a/O/>,,<^xa8>

VC$COMPOSE_KEY  <^a/Y/>,<^a/"/>,,<^xdd>
VC$COMPOSE_KEY  <^a/Y/>,<^a/-/>,,<^xa5>
VC$COMPOSE_KEY  <^a/Y/>,<^a/=/>,,<^xa5>
VC$COMPOSE_KEY  <^a/Y/>,<^x80>,,<^xdd>
```

```
;
; [ \ ] ^ _ `
;
        VC$COMPOSE_KEY  <^a/^/>,<^a/ />,<^>
        VC$COMPOSE_KEY  <^a/^/>,<^a/./>,,<^xb7>
        VC$COMPOSE_KEY  <^a/^/>,<^a\/\>,,<^a/|/>
        VC$COMPOSE_KEY  <^a/^/>,<^a/0/>,,<^xb0>
        VC$COMPOSE_KEY  <^a/^/>,<^a/1/>,,<^xb9>
        VC$COMPOSE_KEY  <^a/^/>,<^a/2/>,,<^xb2>
        VC$COMPOSE_KEY  <^a/^/>,<^a/3/>,,<^xb3>
        VC$COMPOSE_KEY  <^a/^/>,<^a/A/>,,<^xc2>
        VC$COMPOSE_KEY  <^a/^/>,<^a/E/>,,<^xca>
        VC$COMPOSE_KEY  <^a/^/>,<^a/I/>,,<^xce>
        VC$COMPOSE_KEY  <^a/^/>,<^a/O/>,,<^xd4>
        VC$COMPOSE_KEY  <^a/^/>,<^a/U/>,,<^xdb>
        VC$COMPOSE_KEY  <^a/^/>,<^a/a/>,,<^xe2>
        VC$COMPOSE_KEY  <^a/^/>,<^a/e/>,,<^xea>
        VC$COMPOSE_KEY  <^a/^/>,<^a/i/>,,<^xee>
        VC$COMPOSE_KEY  <^a/^/>,<^a/o/>,,<^xf4>
        VC$COMPOSE_KEY  <^a/^/>,<^a/u/>,,<^xfb>

        VC$COMPOSE_KEY  <^a/_/>,<^a/A/>,,<^xaa>
        VC$COMPOSE_KEY  <^a/_/>,<^a/O/>,,<^xba>
        VC$COMPOSE_KEY  <^a/_/>,<^a/a/>,,<^xaa>
        VC$COMPOSE_KEY  <^a/_/>,<^a/o/>,,<^xba>

        VC$COMPOSE_KEY  <^a/`/>,<^a/A/>,,<^xc0>
        VC$COMPOSE_KEY  <^a/`/>,<^a/E/>,,<^xc8>
        VC$COMPOSE_KEY  <^a/`/>,<^a/I/>,,<^xcc>
        VC$COMPOSE_KEY  <^a/`/>,<^a/O/>,,<^xd2>
        VC$COMPOSE_KEY  <^a/`/>,<^a/U/>,,<^xd9>
        VC$COMPOSE_KEY  <^a/`/>,<^a/a/>,,<^xe0>
        VC$COMPOSE_KEY  <^a/`/>,<^a/e/>,,<^xe8>
        VC$COMPOSE_KEY  <^a/`/>,<^a/i/>,,<^xec>
        VC$COMPOSE_KEY  <^a/`/>,<^a/o/>,,<^xf2>
        VC$COMPOSE_KEY  <^a/`/>,<^a/u/>,,<^xf9>
```

```
;
; a to z
;
        VC$COMPOSE_KEY  <^a/a/>,<^a/"/>,,<^xe4>
        VC$COMPOSE_KEY  <^a/a/>,<^a/'/>,,<^xe1>
        VC$COMPOSE_KEY  <^a/a/>,<^a/*/>,,<^xe5>
        VC$COMPOSE_KEY  <^a/a/>,<^a/^/>,,<^xe2>
        VC$COMPOSE_KEY  <^a/a/>,<^a/_/>,,<^xaa>
        VC$COMPOSE_KEY  <^a/a/>,<^a/`/>,,<^xe0>
        VC$COMPOSE_KEY  <^a/a/>,<^a/a/>,<@>
        VC$COMPOSE_KEY  <^a/a/>,<^a/e/>,,<^xe6>          ; order sensitive
        VC$COMPOSE_KEY  <^a/a/>,<^a/~/>,,<^xe3>
        VC$COMPOSE_KEY  <^a/a/>,<^x80>,,<^xe4>
        VC$COMPOSE_KEY  <^a/a/>,<^xb0>,,<^xe5>

        VC$COMPOSE_KEY  <^a/c/>,<^a/,/>,,<^xe7>
        VC$COMPOSE_KEY  <^a/c/>,<^a\/\>,,<^xa2>
        VC$COMPOSE_KEY  <^a/c/>,<^a/0/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/c/>,<^a/o/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/c/>,<^a/|/>,,<^xa2>

        VC$COMPOSE_KEY  <^a/e/>,<^a/"/>,,<^xeb>
        VC$COMPOSE_KEY  <^a/e/>,<^a/'/>,,<^xe9>
        VC$COMPOSE_KEY  <^a/e/>,<^a/^/>,,<^xea>
        VC$COMPOSE_KEY  <^a/e/>,<^a/`/>,,<^xe8>
        VC$COMPOSE_KEY  <^a/e/>,<^x80>,,<^xeb>

        VC$COMPOSE_KEY  <^a/i/>,<^a/"/>,,<^xef>
        VC$COMPOSE_KEY  <^a/i/>,<^a/'/>,,<^xed>
        VC$COMPOSE_KEY  <^a/i/>,<^a/^/>,,<^xee>
        VC$COMPOSE_KEY  <^a/i/>,<^a/`/>,,<^xec>
        VC$COMPOSE_KEY  <^a/i/>,<^x80>,,<^xef>

        VC$COMPOSE_KEY  <^a/l/>,<^a/-/>,,<^xa3>
        VC$COMPOSE_KEY  <^a/l/>,<^a/=/>,,<^xa3>
        VC$COMPOSE_KEY  <^a/l/>,<^a/^/>,,<^xb9>

        VC$COMPOSE_KEY  <^a/n/>,<^a/~/>,,<^xf1>

        VC$COMPOSE_KEY  <^a/o/>,<^a/"/>,,<^xf6>
        VC$COMPOSE_KEY  <^a/o/>,<^a/'/>,,<^xf3>
        VC$COMPOSE_KEY  <^a/o/>,<^a\/\>,,<^xf8>
        VC$COMPOSE_KEY  <^a/o/>,<^a/^/>,,<^xf4>
        VC$COMPOSE_KEY  <^a/o/>,<^a/_/>,,<^xba>
        VC$COMPOSE_KEY  <^a/o/>,<^a/`/>,,<^xf2>
        VC$COMPOSE_KEY  <^a/o/>,<^a/c/>,,<^xa9>
        VC$COMPOSE_KEY  <^a/o/>,<^a/e/>,,<^xf7>          ; order sensitive
        VC$COMPOSE_KEY  <^a/o/>,<^a/s/>,,<^xa7>
        VC$COMPOSE_KEY  <^a/o/>,<^a/x/>,,<^xa8>
        VC$COMPOSE_KEY  <^a/o/>,<^a/~/>,,<^xf5>
        VC$COMPOSE_KEY  <^a/o/>,<^x80>,,<^xf6>
```

```
        VC$COMPOSE_KEY  <^a/p/>,<^a/!/>,,<^xb6>

        VC$COMPOSE_KEY  <^a/s/>,<^a/!/>,,<^xa7>
        VC$COMPOSE_KEY  <^a/s/>,<^a/O/>,,<^xa7>
        VC$COMPOSE_KEY  <^a/s/>,<^a/o/>,,<^xa7>
        VC$COMPOSE_KEY  <^a/s/>,<^a/s/>,,<^xdf>

        VC$COMPOSE_KEY  <^a/u/>,<^a/"/>,,<^xfc>
        VC$COMPOSE_KEY  <^a/u/>,<^a/'/>,,<^xfa>
        VC$COMPOSE_KEY  <^a/u/>,<^a/^/>,,<^xfb>
        VC$COMPOSE_KEY  <^a/u/>,<^a/`/>,,<^xf9>
        VC$COMPOSE_KEY  <^a/u/>,<^x80>,,<^xfc>

        VC$COMPOSE_KEY  <^a/x/>,<^a/O/>,,<^xa8>
        VC$COMPOSE_KEY  <^a/x/>,<^a/o/>,,<^xa8>

        VC$COMPOSE_KEY  <^a/y/>,<^a/"/>,,<^xfd>
        VC$COMPOSE_KEY  <^a/y/>,<^a/-/>,,<^xa5>
        VC$COMPOSE_KEY  <^a/y/>,<^a/=/>,,<^xa5>
        VC$COMPOSE_KEY  <^a/y/>,<^x80>,,<^xfd>

;
; { | } ~
;
        VC$COMPOSE_KEY  <^a/|/>,<^a/C/>,,<^xa2>
        VC$COMPOSE_KEY  <^a/|/>,<^a/c/>,,<^xa2>

        VC$COMPOSE_KEY  <^a/~/>,<^a/ />,,<^a/~/>
        VC$COMPOSE_KEY  <^a/~/>,<^a/A/>,,<^xc3>
        VC$COMPOSE_KEY  <^a/~/>,<^a/N/>,,<^xd1>
        VC$COMPOSE_KEY  <^a/~/>,<^a/O/>,,<^xd5>
        VC$COMPOSE_KEY  <^a/~/>,<^a/a/>,,<^xe3>
        VC$COMPOSE_KEY  <^a/~/>,<^a/n/>,,<^xf1>
        VC$COMPOSE_KEY  <^a/~/>,<^a/o/>,,<^xf5>

;
; Diaeresis mark
;
        VC$COMPOSE_KEY  <^x80>,<^a/A/>,,<^xc4>
        VC$COMPOSE_KEY  <^x80>,<^a/E/>,,<^xcb>
        VC$COMPOSE_KEY  <^x80>,<^a/I/>,,<^xcf>
        VC$COMPOSE_KEY  <^x80>,<^a/O/>,,<^xd6>
        VC$COMPOSE_KEY  <^x80>,<^a/U/>,,<^xdc>
        VC$COMPOSE_KEY  <^x80>,<^a/Y/>,,<^xdd>
        VC$COMPOSE_KEY  <^x80>,<^a/a/>,,<^xe4>
        VC$COMPOSE_KEY  <^x80>,<^a/e/>,,<^xeb>
        VC$COMPOSE_KEY  <^x80>,<^a/i/>,,<^xef>
        VC$COMPOSE_KEY  <^x80>,<^a/o/>,,<^xf6>
        VC$COMPOSE_KEY  <^x80>,<^a/u/>,,<^xfc>
        VC$COMPOSE_KEY  <^x80>,<^a/y/>,,<^xfd>
```

G–8 Three-Stroke Compose Table Default Values

```
;
; Degree sign
;
        VC$COMPOSE_KEY   <^xb0>,<^a/ />,,<^xb0>
        VC$COMPOSE_KEY   <^xb0>,<^a/A/>,,<^xc5>
        VC$COMPOSE_KEY   <^xb0>,<^a/a/>,,<^xe5>

        VC$COMPOSE_KEYEND                          ; END THE TABLE
```

Appendix H

$QIO System Service Description

$QIO System Service Description

$QIO System Service

The Queue I/O Request service queues an I/O request to a channel associated with a device.

The $QIO service completes asynchronously; that is, it returns to the caller immediately after queuing the I/O request, without waiting for the I/O operation to complete.

For synchronous completion, use the Queue I/O Request and Wait ($QIOW) service. The $QIOW service is identical to the $QIO service in every way except that $QIOW returns to the caller after the I/O operation has completed.

Format

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

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. All system services return (by value) a condition value in R0. Condition values that can be returned by this service are listed under "CONDITION VALUES RETURNED."

Arguments

efn

VMS Usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Event flag that $QIO is to set when the I/O operation actually completes. The **efn** argument is a longword value containing the number of the event flag.

If **efn** is not specified, event flag 0 is set.

When $QIO begins execution, it clears the specified event flag or event flag 0 if **efn** was not specified.

The specified event flag is set if the service terminates without queuing an I/O request.

chan

VMS Usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

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

func

VMS Usage: **function_code**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

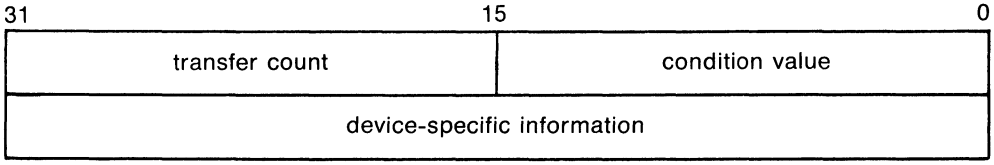
Device-specific function codes and function modifiers specifying the operation to be performed. The **func** is a longword value containing the function code.

Each device has its own function codes and function modifiers. Refer to Chapter 3 and Chapter 4 for complete information about the function codes and function modifiers that apply to the particular I/O operation you want to perform.

iosb

VMS Usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block to receive the final completion status of the I/O operation. The **iosb** is the address of the quadword I/O status block. The following diagram depicts the structure of the I/O status block:



ZK-1723-84

I/O Status Block Fields

condition value
Word-length condition value returned by $QIO when the I/O operation actually completes.

transfer count
Number of bytes of data actually transferred in the I/O operation.

device-specific information
The contents of this field vary depending on the specific device and on the specified function code.

When $QIO begins execution, it clears the quadword I/O status block if the **iosb** argument is specified.

Though this argument is optional, DIGITAL strongly recommends that you specify it for the following reasons.

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the $SYNCH service to synchronize completion of the service, the I/O status block is a required argument for $SYNCH.

- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the $QIO service. The condition value returned in R0 gives you information about the success or failure of the service call itself; the condition value returned in the I/O status block gives you information about the success or failure of the service operation. Therefore, to accurately assess the success or failure of the call to $QIO, you must check the condition values returned in both R0 and the I/O status block.

astadr

VMS Usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

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

The AST routine executes at the access mode of the caller of $QIO.

astprm

VMS Usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

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

p1 to p6

VMS Usage: **varying_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Optional device- and function-specific I/O request parameters. For more information on these parameters see the individual QIO descriptions contained in Chapters 3 and 4.

Description

$QIO operates only on assigned I/O channels and only from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

$QIO uses the following system resources:

- The process' quota for buffered I/O limit (BIOLM) or direct I/O limit (DIOLM).
- The process' buffered I/O byte count (BYTLM) quota.
- The process' AST limit (ASTLM) quota, if an AST service routine is specified.
- System dynamic memory is required to construct a data base to queue the I/O request.
- Additional memory may be required on a device-dependent basis.

For $QIO, completion can be synchronized by (1) specifying the **astadr** argument to have an AST routine execute when the I/O completes or (2) by calling the Synchronize ($SYNCH) service to await completion of the I/O operation. $QIOW completes synchronously, and it is the best choice when synchronous completion is required.

Return Values

SS$_NORMAL	Service successfully completed. The I/O request was successfully queued.
SS$_ABORT	A network logical link was broken.
SS$_ACCVIO	Either the I/O status block cannot be written by the caller, or the parameters for device-dependent function codes are incorrectly specified.
SS$_DEVOFFLINE	The specified device is offline and not currently available for use.
SS$_EXQUOTA	The process has (1) exceeded its AST limit (ASTLM) quota, (2) exceeded its buffered I/O byte count (BYTLM) quota, (3) exceeded its buffered I/O limit (BIOLM) quota, (4) exceeded its direct I/O limit (DIOLM) quota, or (5) requested a buffered I/O transfer smaller than the buffered byte count quota limit (BYTLM), but when added to other current buffer requests, the buffered I/O byte count quota was exceeded.

SS$_ILLEFC	An illegal event flag number was specified.
SS$_INSFMEM	Insufficient system dynamic memory is available to complete the service.
SS$_IVCHAN	An invalid channel number was specified, that is, a channel number of 0 or a number larger than the number of channels available.
SS$_NOPRIV	The specified channel does not exist, was assigned from a more privileged access mode, or the process does not have the necessary privileges to perform the specified functions on the device associated with the specified channel.
SS$_UNASEFC	The process is not associated with the cluster containing the specified event flag.
SS$_LINKABORT	The network partner task aborted the logical link.
SS$_LINKDISCON	The network partner task disconnected the logical link.
SS$_PATHLOST	The path to the network partner task node was lost.
SS$_PROTOCOL	A network protocol error occurred. This is most likely due to a network software error.
SS$_CONNECFAIL	The connection to a network object timed out or failed.
SS$_FILALRACC	A logical link is already accessed on the channel (that is, a previous connect on the channel).
SS$_INVLOGIN	The access control information was found to be invalid at the remote node.
SS$_IVDEVNAM	The NCB has an invalid format or content.
SS$_LINKEXIT	The network partner task was started, but exited before confirming the logical link (that is, $ASSIGN to SYS$NET).
SS$_NOLINKS	No logical links are available. The maximum number of logical links as set for the executor MAXIMUM LINKS parameter was exceeded.
SS$_NOSUCHNODE	The specified node is unknown.
SS$_NOSUCHOBJ	The network object number is unknown at the remote node; or for a TASK = connect, the named DCL command procedure file cannot be found at the remote node.
SS$_NOSUCHUSER	The remote node could not recognize the login information supplied with the connection request.

SS$_PROTOCOL	A network protocol error occurred. This error is most likely due to a network software error.
SS$_REJECT	The network object rejected the connection.
SS$_REMRSRC	The link could not be established because system resources at the remote node were insufficient.
SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_THIRDPARTY	The logical link was terminated by a third party (for example, the System Manager).
SS$_TOOMUCHDATA	The task specified too much optional or interrupt data.
SS$_UNREACHABLE	The remote node is currently unreachable.

Condition Values Returned in the I/O Status Block

Device-specific condition values.

Appendix I

DEC Multinational Character Set

The table below represents the ASCII character set (characters with decimal values 0 through 127). The first half of each of the numbered columns identifies the character as you would enter it on a VT200 or VT100 series terminal or as you would see it on a printer (except for the nonprintable characters). The remaining half of each column identifies the character by the binary value of the byte; the value is stated in three radixes—octal, decimal, and hexadecimal. For example, the letter uppercase A has, under ASCII conventions, a storage value of hexadecimal 41 (a bit configuration of 01000001), equivalent to 101 in octal notation and 65 in decimal notation.

I-2 DEC Multinational Character Set

ROW	b8 b7 b6 b5 b4 b3 b2 b1	0		1		2		3		4		5		6		7	
	COLUMN	0		1		2		3		4		5		6		7	
	BITS	0 0 0		0 0 0		0 0 1		0 0 1		0 1 0		0 1 0		0 1 1		0 1 1	
		0		1		0		1		0		1		0		1	
0	0 0 0 0	NUL	0 0 0	DLE	20 16 10	SP	40 32 20	0	60 48 30	@	100 64 40	P	120 80 50	`	140 96 60	p	160 112 70
1	0 0 0 1	SOH	1 1 1	DC1 (XON)	21 17 11	!	41 33 21	1	61 49 31	A	101 65 41	Q	121 81 51	a	141 97 61	q	161 113 71
2	0 0 1 0	STX	2 2 2	DC2	22 18 12	"	42 34 22	2	62 50 32	B	102 66 42	R	122 82 52	b	142 98 62	r	162 114 72
3	0 0 1 1	ETX	3 3 3	DC3 (XOFF)	23 19 13	#	43 35 23	3	63 51 33	C	103 67 43	S	123 83 53	c	143 99 63	s	163 115 73
4	0 1 0 0	EOT	4 4 4	DC4	24 20 14	$	44 36 24	4	64 52 34	D	104 68 44	T	124 84 54	d	144 100 64	t	164 116 74
5	0 1 0 1	ENQ	5 5 5	NAK	25 21 15	%	45 37 25	5	65 53 35	E	105 69 45	U	125 85 55	e	145 101 65	u	165 117 75
6	0 1 1 0	ACK	6 6 6	SYN	26 22 16	&	46 38 26	6	66 54 36	F	106 70 46	V	126 86 56	f	146 102 66	v	166 118 76
7	0 1 1 1	BEL	7 7 7	ETB	27 23 17	'	47 39 27	7	67 55 37	G	107 71 47	W	127 87 57	g	147 103 67	w	167 119 77
8	1 0 0 0	BS	10 8 8	CAN	30 24 18	(	50 40 28	8	70 56 38	H	110 72 48	X	130 88 58	h	150 104 68	x	170 120 78
9	1 0 0 1	HT	11 9 9	EM	31 25 19	)	51 41 29	9	71 57 39	I	111 73 49	Y	131 89 59	i	151 105 69	y	171 121 79
10	1 0 1 0	LF	12 10 A	SUB	32 26 1A	*	52 42 2A	:	72 58 3A	J	112 74 4A	Z	132 90 5A	j	152 106 6A	z	172 122 7A
11	1 0 1 1	VT	13 11 B	ESC	33 27 1B	+	53 43 2B	;	73 59 3B	K	113 75 4B	[	133 91 5B	k	153 107 6B	{	173 123 7B
12	1 1 0 0	FF	14 12 C	FS	34 28 1C	,	54 44 2C	<	74 60 3C	L	114 76 4C	\	134 92 5C	l	154 108 6C	|	174 124 7C
13	1 1 0 1	CR	15 13 D	GS	35 29 1D	-	55 45 2D	=	75 61 3D	M	115 77 4D	]	135 93 5D	m	155 109 6D	}	175 125 7D
14	1 1 1 0	SO	16 14 E	RS	36 30 1E	.	56 46 2E	>	76 62 3E	N	116 78 4E	^	136 94 5E	n	156 110 6E	~	176 126 7E
15	1 1 1 1	SI	17 15 F	US	37 31 1F	/	57 47 2F	?	77 63 3F	O	117 79 4F	_	137 95 5F	o	157 111 6F	DEL	177 127 7F

KEY

CHARACTER		
ESC	33	OCTAL
	27	DECIMAL
	1B	HEX

ZK-1752-84

The ASCII character set comprises the first half of the DEC multinational character set. The following table represents the second half of the DEC multinational character set (characters with decimal values 128 through 255). The first half of each of the numbered columns identifies the character as you would see it on a VT200 series terminal or printer (these characters cannot be output on a VT100 series terminal).

I-4 DEC Multinational Character Set

8		9		10		11		12		13		14		15		COLUMN	
1 0 0 0		1 0 0 1		1 0 1 0		1 0 1 1		1 1 0 0		1 1 0 1		1 1 1 0		1 1 1 1		b8 b7 BITS b6 b5 b4 b3 b2 b1	ROW
	200 128 80	DCS	220 144 90		240 160 A0	º	260 176 B0	À	300 192 C0		320 208 D0	à	340 224 E0		360 240 F0	0 0 0 0	0
	201 129 81	PU1	221 145 91	¡	241 161 A1	±	261 177 B1	Á	301 193 C1	Ñ	321 209 D1	á	341 225 E1	ñ	361 241 F1	0 0 0 1	1
	202 130 82	PU2	222 146 92	¢	242 162 A2	2	262 178 B2	Â	302 194 C2	Ò	322 210 D2	â	342 226 E2	ò	362 242 F2	0 0 1 0	2
	203 131 83	STS	223 147 93	£	243 163 A3	3	263 179 B3	Ã	303 195 C3	Ó	323 211 D3	ã	343 227 E3	ó	363 243 F3	0 0 1 1	3
IND	204 132 84	CCH	224 148 94		244 164 A4		264 180 B4	Ä	304 196 C4	Ô	324 212 D4	ä	344 228 E4	ô	364 244 F4	0 1 0 0	4
NEL	205 133 85	MW	225 149 95	¥	245 165 A5	µ	265 181 B5	Å	305 197 C5	Õ	325 213 D5	å	345 229 E5	õ	365 245 F5	0 1 0 1	5
SSA	206 134 86	SPA	226 150 96		246 166 A6	¶	266 182 B6	Æ	306 198 C6	Ö	326 214 D6	æ	346 230 E6	ö	366 246 F6	0 1 1 0	6
ESA	207 135 87	EPA	227 151 97	§	247 167 A7	·	267 183 B7	Ç	307 199 C7	Œ	327 215 D7	ç	347 231 E7	œ	367 247 F7	0 1 1 1	7
HTS	210 136 88		230 152 98	¤	250 168 A8		270 184 B8	È	310 200 C8	Ø	330 216 D8	è	350 232 E8	ø	370 248 F8	1 0 0 0	8
HTJ	211 137 89		231 153 99	©	251 169 A9	1	271 185 B9	É	311 201 C9	Ù	331 217 D9	é	351 233 E9	ù	371 249 F9	1 0 0 1	9
VTS	212 138 8A		232 154 9A	a	252 170 AA	o	272 186 BA	Ê	312 202 CA	Ú	332 218 DA	ê	352 234 EA	ú	372 250 FA	1 0 1 0	10
PLD	213 139 8B	CSI	233 155 9B	«	253 171 AB	»	273 187 BB	Ë	313 203 CB	Û	333 219 DB	ë	353 235 EB	û	373 251 FB	1 0 1 1	11
PLU	214 140 8C	ST	234 156 9C		254 172 AC	¼	274 188 BC	Ì	314 204 CC	Ü	334 220 DC	ì	354 236 EC	ü	374 252 FC	1 1 0 0	12
RI	215 141 8D	OSC	235 157 9D		255 173 AD	½	275 189 BD	Í	315 205 CD	Ý	335 221 DD	í	355 237 ED	ý	375 253 FD	1 1 0 1	13
SS2	216 142 8E	PM	236 158 9E		256 174 AE		276 190 BE	Î	316 206 CE		336 222 DE	î	356 238 EE		376 254 FE	1 1 1 0	14
SS3	217 143 8F	APC	237 159 9F		257 175 AF	¿	277 191 BF	Ï	317 207 CF	ß	337 223 DF	ï	357 239 EF		377 255 FF	1 1 1 1	15

KEY

CHARACTER: ESC

33 OCTAL
27 DECIMAL
1B HEX

ZK-1753-84

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

MicroVMS Workstation
Video Device Driver Manual
AA–DY65D–TE
AD–DY65D–T1
AD–DY65D–T2

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

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
MLO5–5/E45
146 MAIN STREET
MAYNARD, MA 01754–2571

Do Not Tear — Fold Here

Cut Along Dotted Line