

VMS Workstation Software Video Device Driver Manual

Order Number: AA-DY65E-TE

June 1989

The *VMS Workstation Software Video Device Driver Manual* provides technical information about the QVSS and QDSS drivers.

This manual contains Update Notice 1, AD-DY65E-T1.

Operating System and Version: VMS Version 5.0 and later

Software Version: VMS Workstation Software Version 4.1

Digital Equipment Corporation

June 1989

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

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.


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

Copyright ©1989 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	

This document was prepared using VAX DOCUMENT, Version 1.1

INSTRUCTIONS

The enclosed pages are to be placed in the *VMS Workstation Software Video Device Driver Manual* as replacements for or additions to the current pages. 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.

Copyright Digital Equipment Corporation 1989
All Rights Reserved.
Printed in U.S.A.

Old Page(s)	New Page(s)
Title/Copyright	Title/Copyright
iii/iv through xi/xii	iii/iv through xi/xii
1-1/1-2 through 1-3/1-4	1-1/1-2 through 1-4.1/blank
1-9/1-10 through 1-11/1-12	1-9/1-10 through 1-12.1/blank
2-15/2-16	2-15/2-16
2-19/2-20 through 2-23/24	2-19/2-20 through 2-24.1/blank
Chapter 3	Chapter 3
4-7/4-8	4-7/4-8
4-13/4-14	4-13/4-14
4-23/4-24	4-23/4-24
5-7/5-8 through 5-9/5-10	5-7/5-8 through 5-9/5-10
5-17/5-18 through 5-21/5-22	5-17/5-18 through 5-21/5-22
5-69/5-70 through 5-71/5-72	5-69/5-70 through 5-71/5-72
Appendix A	Appendix A
Appendix B	Appendix B
Appendix C	Appendix C
Index-1/Index-2 through Index-5/Index-6	Index-1/Index-2 through Index-5/Index- 6
Reader's Comments/Mailer	

Contents

PREFACE		xi
<hr/>		
CHAPTER 1	VIDEO DEVICE DRIVER INTRODUCTION	1-1
<hr/>		
1.1	OVERVIEW OF THE DRIVERS	1-1
1.1.1	Driver Differences	1-2
<hr/>		
1.2	CHOOSING AN INTERFACE	1-3
<hr/>		
1.3	HOW THE DRIVERS WORK	1-4
1.3.1	Defining Regions on the Screen	1-4
1.3.2	Driver Communication Mechanisms	1-4.1
1.3.2.1	QIO Interface • 1-5	
1.3.2.2	Request Queue Interface • 1-6	
1.3.3	How Drivers Use Memory	1-6
1.3.3.1	How QVSS Uses Video Memory • 1-7	
1.3.3.2	How QDSS Uses Video Memory • 1-9	
1.3.4	How the Drivers Track Screen Events	1-12
1.3.5	Cursor Pattern List	1-12
1.3.6	Pointer Button Transition List	1-13
1.3.7	Pointer Movement List	1-13
1.3.8	Keyboard Entry List	1-14
1.3.9	Occlusion	1-15
1.3.10	Viewports	1-15
1.3.10.1	Viewport Update Regions • 1-15	
1.3.10.2	Using Update Regions for Occlusion • 1-16	
1.3.11	QDSS Drawing Operation Queues	1-16
1.3.11.1	Request Queue • 1-16	
1.3.11.2	Return Queue • 1-17	
1.3.12	Deferred Queue	1-18
<hr/>		
CHAPTER 2	PROGRAMMING TO THE DRIVER	2-1
<hr/>		
2.1	INITIALIZING THE SCREEN	2-1
<hr/>		
2.2	ACCESSING THE SYSTEM INFORMATION BLOCK	2-1

Contents

2.3	USING CHANNELS WITH VIDEO DEVICE DRIVERS	2-2
2.4	USING THE KEYBOARD	2-3
2.4.1	Receiving Keyboard Input _____	2-3
2.4.2	Keyboard Characteristics _____	2-6
2.4.3	Modifying the Keyboard Table _____	2-7
2.4.3.1	Constructing a Keyboard Table with Macros • 2-7	
2.4.3.2	Constructing a Keyboard Table Without Macros • 2-9	
2.4.3.3	Loading a Keyboard Table • 2-11	
2.4.4	Composing Nonstandard Characters _____	2-11
2.4.5	Constructing Compose Sequence Tables _____	2-12
2.4.5.1	Using Macros to Construct Compose Sequence Tables • 2-15	
2.4.5.2	Loading a Compose Table • 2-16	
2.5	USING A POINTER DEVICE	2-17
2.6	USING THE TYPE-AHEAD BUFFER	2-20
2.6.1	Getting Input from the Type-Ahead Buffer _____	2-20
2.7	INTERCEPTING INPUT	2-21
2.8	DEFINING CURSOR PATTERNS	2-21
2.8.1	Multiplane Cursor Patterns _____	2-22
2.9	USING AN ALTERNATE WINDOWING SYSTEM	2-24
2.10	DRAWING TO THE QVSS SCREEN	2-24
2.10.1	Manipulating Bits in Video Memory _____	2-24
2.10.2	Mapping Video Memory to the Screen _____	2-24.1
2.11	CREATING A QDSS VIEWPORT	2-25
2.11.1	Assigning a Viewport Channel _____	2-25
2.11.2	Getting a Viewport ID _____	2-25
2.11.3	Defining a Viewport _____	2-25
2.11.4	Starting the Viewport _____	2-26
2.12	DRAWING WITH THE QDSS DRIVER	2-28

2.13	USING BITMAPS	2-28
2.14	SYNCHRONIZING VIEWPORT ACTIVITY	2-29
2.15	HANDLING OCCLUSION	2-31
2.15.1	Redefining Viewports	2-31
2.15.2	Securing Exclusive Access to the Bitmap	2-32
2.15.3	Popping an Occluded Viewport	2-36
2.16	DELETING A VIEWPORT	2-40
2.16.1	Synchronizing Viewport Deletion	2-41
2.16.2	Erasing a Viewport	2-41
2.17	MOVING A VIEWPORT	2-45
2.18	USING THE DEFERRED QUEUE	2-45
2.19	USING COLOR	2-46
2.19.1	Informing the Driver About Color	2-47
2.19.2	Manipulating Color Map Values	2-47
CHAPTER 3 QVSS/QDSS COMMON QIO INTERFACE		3-1
3.1	HOW TO USE THIS CHAPTER	3-1
3.1.1	QIO Description Format	3-2
	DEFINE POINTER CURSOR PATTERN	3-3
	ENABLE BUTTON TRANSITION	3-8
	ENABLE DATA DIGITIZING	3-13
	ENABLE FUNCTION KEYS	3-17
	ENABLE INPUT SIMULATION	3-20
	ENABLE KEYBOARD INPUT	3-23
	ENABLE KEYBOARD SOUND	3-29
	ENABLE POINTER MOVEMENT	3-31
	ENABLE USER ENTRY	3-35
	GET KEYBOARD CHARACTERISTICS	3-37
	GET NEXT INPUT TOKEN	3-39
	GET NUMBER OF LIST ENTRIES	3-40
	GET SYSTEM INFORMATION	3-41
	INITIALIZE SCREEN	3-42
	LOAD COMPOSE SEQUENCE TABLE	3-43
	LOAD KEYBOARD TABLE	3-45
	MODIFY KEYBOARD CHARACTERISTICS	3-47

Contents

MODIFY SYSTEMWIDE CHARACTERISTICS	3-51
REVERT TO DEFAULT COMPOSE TABLE	3-55
REVERT TO DEFAULT KEYBOARD TABLE	3-56

CHAPTER 4 QDSS-SPECIFIC QIO INTERFACE 4-1

4.1 HOW TO USE THIS CHAPTER 4-1

4.2 QIO DESCRIPTION FORMAT 4-2

DEFINE VIEWPORT REGION	4-3
DELETE DEFERRED QUEUE OPERATION	4-5
EXECUTE DEFERRED QUEUE	4-6
GET COLOR MAP ENTRIES	4-7
GET FREE DOPS	4-9
GET VIEWPORT ID	4-10
HOLD VIEWPORT ACTIVITY	4-12
INSERT DOP	4-13
LOAD BITMAP	4-14
NOTIFY DEFERRED QUEUE FULL	4-18
READ BITMAP	4-19
RELEASE HOLD	4-22
RESUME VIEWPORT ACTIVITY	4-23
SET COLOR CHARACTERISTICS	4-23
SET COLOR MAP ENTRIES	4-24
START REQUEST QUEUE	4-26
STOP REQUEST QUEUE	4-27
SUSPEND OCCLUDED VIEWPORT ACTIVITY	4-28
SUSPEND VIEWPORT ACTIVITY	4-29
WRITE BITMAP	4-30

CHAPTER 5 USING DRAWING OPERATION PRIMITIVES 5-1

5.1 OVERVIEW OF DOPS 5-1

5.2 DOP STRUCTURE 5-2

5.3 IMPLEMENTING DOPS IN THE UIS ENVIRONMENT 5-3

5.3.1	Allocating Storage for DOPs in the UIS Environment	5-3
5.3.1.1	Allocation Mechanism • 5-3	
5.3.1.2	Modifying DOP Size and Number • 5-4	
5.3.2	Executing DOPs in the UIS Environment	5-5
5.3.2.1	Execution Mechanism • 5-6	

5.4	IMPLEMENTING DOPS IN A NON-UIS ENVIRONMENT	5-6
5.4.1	Allocating Storage for DOPS in a Non-UIS Environment	5-7
5.4.2	Executing a DOP in a Non-UIS Environment	5-9
<hr/>		
5.5	STRUCTURING AND INITIALIZING DOPS	5-9
5.5.1	Common Block	5-10
5.5.2	Unique Block	5-12
5.5.3	Variable Block	5-12
5.5.4	Programming Considerations	5-12
	5.5.4.1 The Predefined DOP Structure • 5-12	
	5.5.4.2 Using the Examples • 5-13	
<hr/>		
5.6	THE DOP REFERENCE	5-15
	COMMON BLOCK	5-16
	DELETE BITMAP	5-23
	DRAW COMPLEX LINE	5-24
	DRAW FIXED TEXT	5-28
	DRAW LINES	5-31
	DRAW POINTS	5-34
	DRAW VARIABLE TEXT	5-37
	FILL LINES	5-41
	FILL POINT	5-44
	FILL POLYGON	5-48
	MOVE AREA	5-52
	MOVE/ROTATE AREA	5-56
	RESUME VIEWPORT ACTIVITY	5-61
	SCROLL AREA	5-63
	START REQUEST QUEUE	5-67
	STOP REQUEST QUEUE	5-69
	SUSPEND VIEWPORT ACTIVITY	5-70
<hr/>		
5.7	UISDC DOP INTERFACE	5-72
5.7.1	Loading Bitmaps into Offscreen Memory	5-72
	UISDC\$ALLOCATE_DOP	5-74
	UISDC\$LOAD_BITMAP	5-76
	UISDC\$EXECUTE_DOP_ASYNCH	5-78
	UISDC\$EXECUTE_DOP_SYNCH	5-80
	UISDC\$QUEUE_DOP	5-81

APPENDIX A QVSS/QDSS DATA STRUCTURES

A-1

Contents

APPENDIX B	QDSS-SPECIFIC DATA STRUCTURES	B-1
-------------------	--------------------------------------	------------

APPENDIX C	QDSS WRITING MODES	C-1
-------------------	---------------------------	------------

APPENDIX D	QVSS PROGRAMMING EXAMPLE	D-1
-------------------	---------------------------------	------------

D.1	PROGRAMMING	D-1
D.1.1	Program Functions _____	D-1
D.1.2	QVSS Program Example _____	D-2

APPENDIX E	KEYBOARD TABLE MACRO	E-1
-------------------	-----------------------------	------------

APPENDIX F	COMPOSE TABLE MACROS	F-1
-------------------	-----------------------------	------------

APPENDIX G	DEFAULT THREE-STROKE COMPOSE TABLE VALUES	G-1
-------------------	--	------------

APPENDIX H	\$QIO SYSTEM SERVICE DESCRIPTION	H-1
	\$QIO SYSTEM SERVICE DESCRIPTION	H-2

APPENDIX I	DEC MULTINATIONAL CHARACTER SET	I-1
-------------------	--	------------

APPENDIX J	ISO LATIN NR 1 SUPPLEMENTAL CHARACTER SET	J-1
-------------------	--	------------

INDEX		
--------------	--	--

EXAMPLES

2-1	Enabling Keyboard Requests _____	2-5
2-2	Modifying the North American Keyboard _____	2-10
2-3	Loading a Keyboard Table _____	2-11
2-4	Loading a Three-Stroke Compose Sequence _____	2-15
2-5	How to Load a Three-Stroke Compose Table _____	2-16
2-6	How to Program a Pointer Motion AST _____	2-18
2-7	Typical Programming and Use of Pointer Button ASTs _____	2-20
2-8	Assignment of a Single-Plane Cursor Region Pattern _____	2-23
2-9	Creating a Viewport _____	2-27
2-10	Securing Bitmap Access _____	2-34
2-11	Popping a Viewport _____	2-37
2-12	Deleting a Viewport _____	2-42
5-1	Allocating a DOP _____	5-4
5-2	Queuing a DOP for Execution _____	5-6
5-3	Allocating a DOP _____	5-7
5-4	Inserting a DOP on the Request Queue _____	5-10
5-5	Calling Program for Example Subroutines _____	5-14

FIGURES

1-1	The Driver Interface _____	1-2
1-2	QVSS Video Memory and Scanline Map _____	1-8
1-3	Scanline Map Mapping Nonconsecutive Memory _____	1-9
1-4	QDSS Video Map _____	1-10
1-5	Cycling the Keyboard List _____	1-14
1-6	Viewport Update Region Data Structure _____	1-16
2-1	Keyboard Table Layout _____	2-8
2-2	Keyboard Table Description _____	2-10
2-3	Three-Stroke Compose Sequence Table Description _____	2-12
2-4	Three-Stroke Compose Sequence Table _____	2-13
2-5	Two-Stroke Compose Sequence Table Description _____	2-14
2-6	Two-Stroke Compose Sequence Table _____	2-15
2-7	Viewport and Update Region Definition Buffer _____	2-26
2-8	Synchronizing Viewport Activity _____	2-29
2-9	Occluded Viewport _____	2-31
2-10	Redefining Viewports with URDs _____	2-32
2-11	Indexing the Hardware Color Map _____	2-48
4-1	Large Font Defined Across Bitmap Blocks _____	4-16
5-1	How the Source Index Works _____	5-20
I-1	DEC Multinational Character Set—I _____	I-2
I-2	DEC Multinational Character Set—II _____	I-3

Contents

TABLES

1-1	Device Driver Differences _____	1-2
2-1	Key States _____	2-8
2-2	Diacritical Characters _____	2-9
2-3	Diacritical Characters _____	2-13
3-1	QIO Functional Groups _____	3-1
4-1	QDSS QIO Functional Groups _____	4-1
5-1	Redefinition of Logical Values _____	5-5
5-2	Symbolic Constants _____	5-11
C-1	QDSS Writing Modes _____	C-1
C-2	QDSS Writing Mode Modifiers _____	C-2

Preface

The *VMS Workstation Software 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 you program to the driver are provided in reference form. Data types used to program to the drivers are illustrated in the reference sections, 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 systems programmers who are writing applications directly to the driver.

Document Structure

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.

Preface

- Appendix J contains the ISO Latin NR 1 supplemental character set.


Associated Documents

The following VMS manuals are related to this manual:

- *VMS Workstation Software Graphics Programming Guide*
- *VMS Workstation Software User's Guide*
- *VMS User's Guide*

Conventions

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.
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  . 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.

1

Video Device Driver Introduction

This chapter introduces the concepts and terms that describe how to write an application to interact with the QVSS and QDSS video device drivers.

This chapter describes the following topics:

- The two available video device drivers (QVSS and QDSS).
- How to determine which programming interface your application should address. (Your application might 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 can have one of the following two video device drivers:

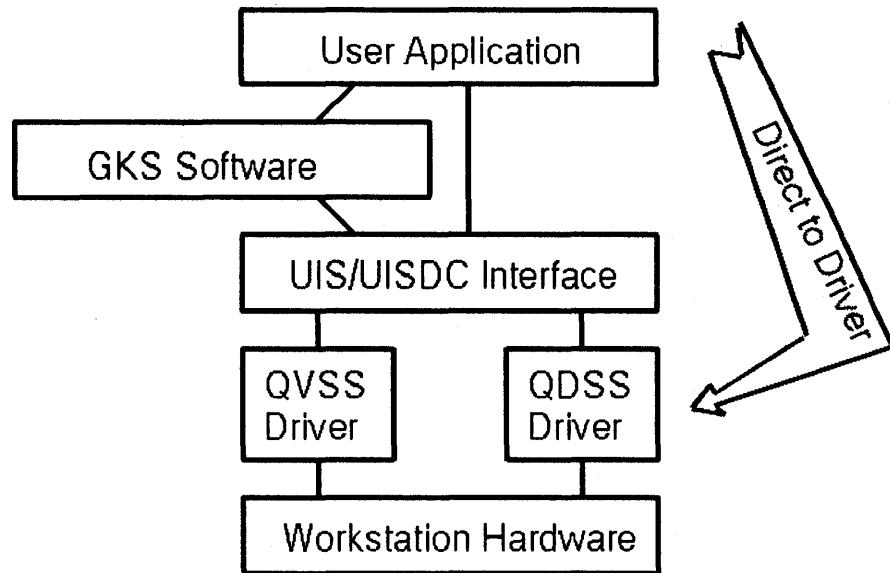
- QVSS-QBus Video Subsystem
- QDSS-QBus Device Subsystem

Although both drivers allow you to create graphics applications with VAXstation features, each driver requires unique hardware. Thus, 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 uniformly. All VAXstation software uses the driver to access hardware, either directly or indirectly. Figure 1-1 illustrates the layered relationship of applications, VAXstation software, device driver, and VAXstation hardware.

Video Device Driver Introduction

Figure 1-1 The Driver Interface



1.1 Driver Differences

The primary differences between the QVSS driver and the QDSS driver are:

- Use of color
- Use of global sections
- Method of bitmap manipulation
- Ability to provide alternate windowing systems

Table 1-1 lists device driver capabilities.

Table 1-1 Device Driver Differences

Feature	Driver	
	QVSS	QDSS
Color	Bitonal	Gray-scale, color
Memory	One-plane	Multiplane
Bit Manipulation	Direct	DOPs
Alternate Windowing	Available	Not available

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 draws color and gray-scale images.

Bitmap Manipulation

The QVSS driver supports only 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 drawing operation primitives (DOPs) that make drawing to the bitmap easier and faster. 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. When you use an alternate windowing system, both windowing systems share video memory.

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

1.2 Choosing an Interface

When you write an application, determine which level of interface your application should address. As shown in Figure 1-1, your application can address the system at the UIS level, the driver level, or the hardware level. For each successive level downward, your application increases its degree of control but increases its difficulty and the amount of work it 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 you write high-level graphics applications. UIS routines use a device-independent world coordinate system.

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

Video Device Driver Introduction

Hardware Interface

You can bypass both the UIS interface and the driver interface and directly address the hardware. If your application does not need a windowing system, 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.

.3 How the Drivers Work

Before you can write to the driver interface, you should understand the concepts and terms explained in this section.

.3.1 Defining Regions on the Screen

Both drivers address rectangular portions of the screen called *regions*. The QDSS driver accesses regions of the screen as *viewports*. Section 1.3.10 describes viewports. Your application uses driver interface QIOs to define addressable screen regions.

For the driver to define a region, your application must associate the region with a unique channel. The channel provides a logical path that connects an application to the device driver. Before you define a region, call the \$ASSIGN system service to obtain a unique channel number for the region. (The *VAX/VMS System Services Reference Manual* has more information on \$ASSIGN.)

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, define the region with the Enable Button Transition QIO. To cause an action to occur when the event is detected in that region, include 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 transition.

The driver detects when an event occurs and ensures that the proper action takes place. Section 1.3.4 describes how the drivers manage regions and events.

1.3.2 **Driver Communication Mechanisms**

An application can use a QIO interface to access either driver. An application can also use a mechanism called the request queue to access the QDSS driver.

1.3.2.1 QIO Interface

The QIO interface is the method of access common to most drivers. The QVSS and QDSS drivers provide a number of QIOs that perform the following driver-specific functions:

- Screen initialization
- Pointer movement region definition
- Bitmap copy performance

While the QIO interface is common to both drivers, some QIOs are QDSS-specific. Any driver QIO can be used with the QDSS driver, (see Chapter 4), while the QIOs that apply to both the QVSS and QDSS drivers are a subset of the entire QIO interface (see Chapter 3).

Most QIOs are input functions. When you call them, you typically specify `IO$_SETMODE` as the function parameter. In these QIOs, the `P1` parameter actually serves as the distinguishing function code.

The remaining QIOs are output functions. When you call them, you typically specify `IO$_SENSEMODE` as the function parameter. In these QIOs, the `P1` parameter actually serves as the distinguishing function code.

Although the QIO interface permits a wide range of functions, some are 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*. To associate regions with events, the QVSS and QDSS drivers maintain one list for each event type. 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 indicated by the appropriate list entry. Section 1.3.4 explains how the driver constructs and manages lists.

Returning System Information

Your application gets information from the system with output QIOs. The Get System Information QIO returns the system information block, which describes the state of the system in:

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

Video Device Driver Introduction

The QDSS system information block contains all the same fields as the QVSS information system block, but includes additional fields. Appendices A and B illustrate and describe the structure of both system blocks:

- QVB—QVSS system information block
- QDB—QDSS system information block

You can also use the Get Keyboard Characteristics QIO to inquire about the characteristics of the current keyboard.

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.

To manipulate the QDSS-specific queues, you use three QDSS-specific output QIOs:

- Request queue
- Return queue
- Deferred queue

These QIOs permit an application to stop and start processing queues. The request queue is briefly described in the next section. A full description of the three queues appears at the end of this chapter.

1.3.2.2 Request Queue Interface

This section applies only to QDSS systems.

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

To operate the request queue, your application uses drawing operation primitives (DOPs), structures that contain 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. DOPs are placed on the queue in execution (drawing) order.

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

1.3.3 How 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. To understand how to program to the drivers, you must understand how the drivers use memory.

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. To map lines of video memory to lines on the screen, QVSS uses a data structure called the *scanline map*.

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 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 display at a time and total video memory is 2048 lines, QVSS video memory is referred to in two sections:

- Onscreen memory—Any portion of video memory currently displayed. Since the largest VAXstation display is 864 lines, that is the maximum size of onscreen memory.
- Offscreen memory—The 1184 undisplayed lines of video memory.

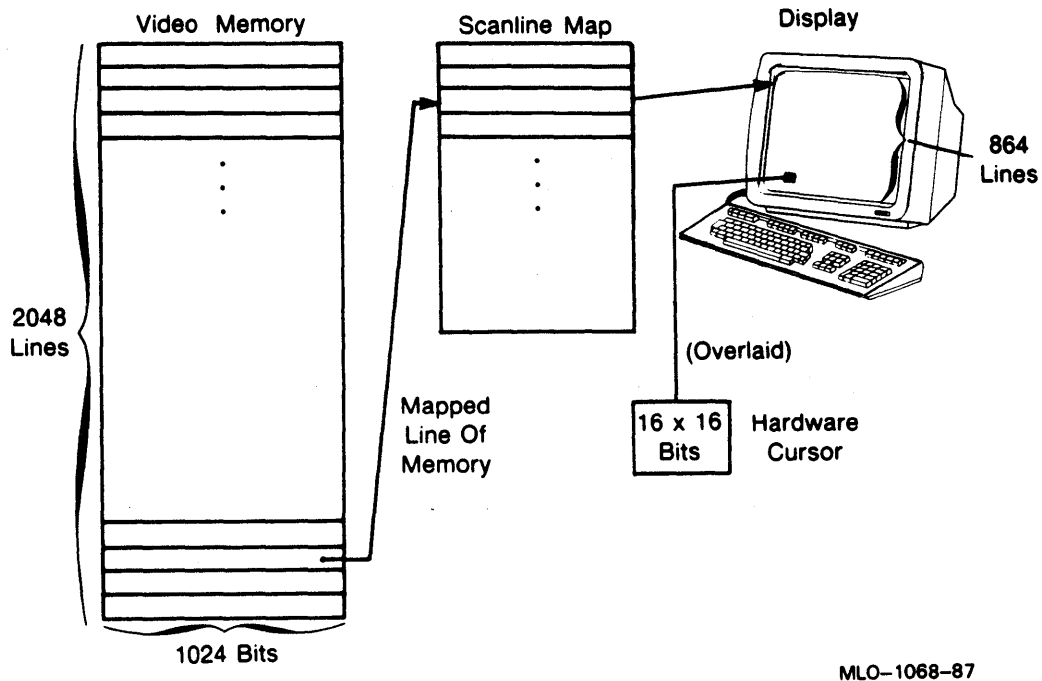
You use offscreen memory to store images or fonts that are not currently being displayed, as well as to handle occlusion (see Section 1.3.9).

NOTE: On a VAXstation 2000 monochrome monitor, 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.

Video Device Driver Introduction

Figure 1-2 QVSS Video Memory and Scanline Map



Hardware Cursor

The screen cursor is defined by a separate block of hardware memory as follows:

- QVSS single-plane cursor systems—16 x 16 bits
- QDSS multiplane cursor systems—16 x 32 bits

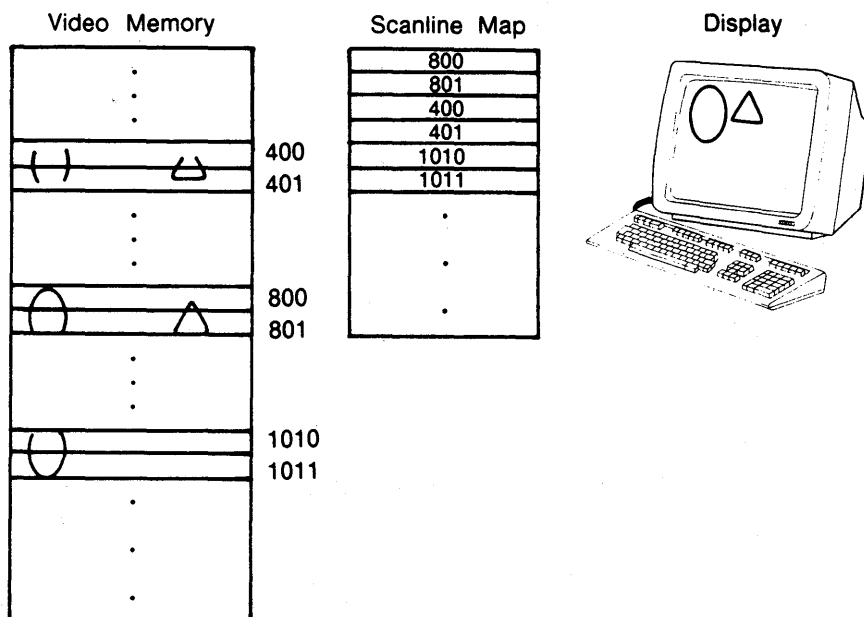
This block stores the bitmap image of the cursor pattern. It is *not* part of video memory.

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.

Scanline Map

Note that the scanline map need not index consecutive lines of video memory. That is, an object can be represented in nonconsecutive lines in video memory (when 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



ZK-5247-86

Accessing QVSS Video Memory

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

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

An application should issue a QVB request for each process. Because the address of the system block does not change, 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 a 1024- by 864-pixel area, or 864 lines that are 1024 pixels in length. The full QDSS video memory is a 2048- 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. 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 called *onscreen memory*.

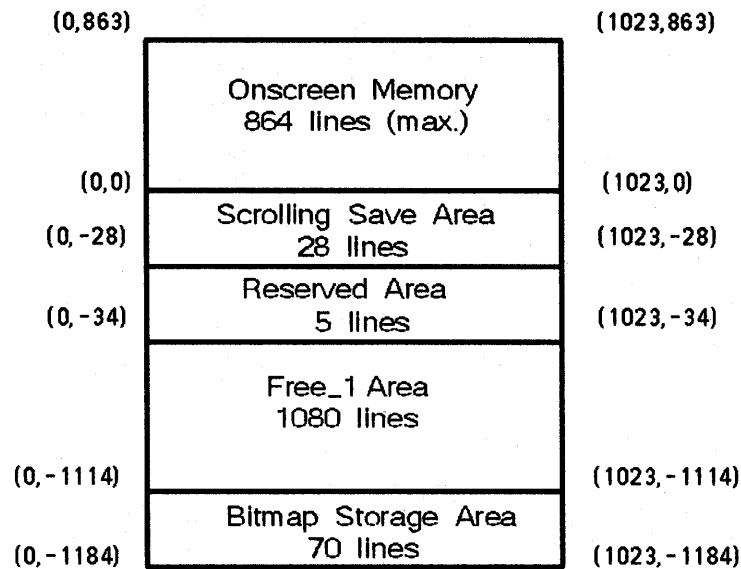
Video Device Driver Introduction

The remaining 1184 lines of video memory are called *offscreen memory*. Offscreen memory handles occlusion. The offscreen portion of the video memory is further divided into the following fixed-length sections:

- Scrolling save area
- Reserved 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 corner of onscreen memory is assigned the coordinate (0,0). This corresponds to the lower left 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.

Bitmap Storage Area

The driver uses this area to store bitmaps:

- Fonts
- Images
- Pattern fills

The area has a 70-line by 1024-bit block of memory for each memory plane on the system. To accommodate more fonts in memory, some planes partition the 70-line blocks into two 35-line sections. An application uses this area to load any defined bitmaps stored in VAX memory.

Use the Load Bitmap QIO or the UISDC\$LOAD_BITMAP routine to load the information in VAX memory into the bitmap storage area.

NOTE: If a bitmap does not fit into the bitmap storage area, the application must partition the information into one 70-line section for a single-plane image or two 35-line sections for multiplane images.

QDSS Video Memory Access

QDSS does *not* permit direct bitmap access. Use the DOP interface to draw to video memory, and use the Read Bitmap and Write Bitmap QIOs to copy images to video memory.

Viewports—The QDSS driver must direct all operations to the screen via 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 might require exclusive bitmap access to video memory. When you perform an exclusive-access operation, no other operation should access onscreen memory. Since the QDSS driver controls onscreen memory access, 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
- Bitmap-to-bitmap

Drivers use the QIO interface to execute transfers, which require exclusive bitmap access for synchronization.

QDB—An application can 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

To track events that occur on the screen, the QVSS and QDSS drivers keep a list for each of the following event types:

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

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

The drivers associate each of the first three events with a specific region; they associate the keyboard entry event 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 information:

- Region definition
- 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; it uses the region definition to identify which list entry AST to issue. For example, if the driver detects a button transition, it searches the button transition list to determine whether the region where the transition occurred has a button transition AST enabled. For a keyboard entry, the first entry on the keyboard list is always invoked.

The following sections summarize how drivers handle each type of event.

1.3.5 Cursor Pattern List

The cursor pattern list determines the pattern of the cursor for a region. Use the Define Pointer Cursor Pattern QIO to define the cursor shape and hot spot for a region. The list entry contains the following information:

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

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

Video Device Driver Introduction

When the driver detects cursor movement, it searches the cursor pattern list for the appropriate ASTs to deliver. If a cursor pattern is to change, 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 Pointer Button Transition List

The driver uses this list to determine what action to take when a pointer button transition occurs. A button transition can be either up or down. It is detected by the hardware.

A button transition event occurs when you press or release the pointer button within a region defined with the Enable Button Transition QIO. Each pointer button transition list entry contains the following information:

- Unique channel ID
- AST address
- 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. The AST routine defined for that region determines what occurs when a pointer button is pressed. A token passes to the specified AST routine to signal the following:

- Which button made a transition
- Type of transition (up or down)

The Enable Button Transition QIO description explains how button transitions are represented.

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

1.3.7 Pointer Movement List

The driver uses the pointer movement list to determine what action to take when the pointing device is moved. Use the Enable Pointer Movement QIO to define a region in which to take special action when pointer movement occurs. Each pointer motion list entry contains the following information:

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

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

If the pointer cursor moves outside a currently active region, a special exit token of -1 passes to the AST to notify it of the occurrence. When the cursor leaves the region, a process might want to perform some cleanup functions.

1.3.8 Keyboard Entry List

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

Each keyboard list entry contains the following information:

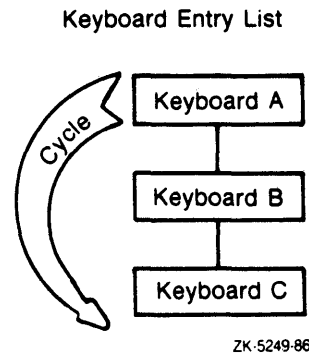
- Unique channel ID
- AST service routine address
- Address of the longword that receives an input token when the AST routine is called

See the Enable Keyboard Input QIO description for more details.

This event differs from others, because when 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.

Popping or cycling operations cause a keyboard to become active. 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.

1.3.9 Occlusion

In a windowing system, more than one window can address the same area of the screen. However, only one window can be displayed in one area at a time. Occlusion occurs when one window display hides some or all of another display.

2.4.5.1 Using Macros to Construct Compose Sequence Tables

To construct a compose table, initialize it, then load the sequences you want to define. VMS Workstation Software provides macros to generate compose tables in SYS\$LIBRARY:VWSSYSDEF.MLB:

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

These macros are also described in Appendix F.

Initializing a Table

Call VC\$COMPOSE_KEYINIT to initialize a table. This macro has two parameters:

- The address of the table, which it *returns* after it allocates space and initializes the table. Specify this parameter and use the returned address when you load 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

Call VC\$COMPOSE_KEY to load a compose sequence. 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. Example 2-4 illustrates loading a three-stroke compose sequence.

Example 2-4 Loading a Three-Stroke Compose Sequence

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

Terminating a Table

Call VC\$COMPOSE_KEYEND to terminate a keyboard table. This macro *returns* one parameter, the length of the table. Typically, you specify this parameter and use the returned length when you load the table.

2.4.5.2 Loading a Compose Table

To load a compose table, use the Load Compose Sequence Table QIO with the address and size of the table and the channel of the keyboard entry with which you want to associate the table. The VC\$COMPOSE_KEYINIT and VC\$COMPOSE_KEYEND macros return address and length, respectively.

VMS Workstation Software is shipped with copies of the Digital standard three-stroke and two-stroke compose tables that reside in the driver. These tables are the default until you load alternates.

NOTE: Digital standard two-stroke compose sequences are not supported on the North American keyboard.

To revert to the default compose table, call the Revert to Default Compose Table QIO.

Example 2-5 illustrates how to load a three-stroke compose table. (Appendix D shows this example in the context of a complete application program.)

Example 2-5 How to Load a Three-Stroke Compose 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$:
    VC$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
.
.
.
```

- Use the MACRO instruction INSQUE

At specific intervals, the driver scans all request queues to check for work. If a queue contains DOPs, the driver removes the DOPs from the queue and performs the specified operations. The packets are stored on the queue in drawing operation order.

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

1.3.11.4 Return Queue

A DOP is a data structure for which storage must be allocated. If you process a large number of DOPs without any restrictions, they might consume all available system memory (or enough to degrade performance considerably).

The QDSS driver provides the return queue 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 called a *free DOP*.

To save space, an application can check the return queue for used DOP storage before it allocates memory for new drawing operations.

Allocating DOPs

To allocate storage for DOPs, use either the UISDC interface (see Chapter 5) or memory allocation system routines. When you allocate a DOP, initialize the DOP size fields of the DOP queue structure and the request queue structure (see Appendix B). You can choose either small or large size.

Before it allocates new storage, an application should check the return queue for reusable DOP 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 how many return queue DOPs to wait for 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 allocation of too much system memory. For example, an application allocates 300 DOPs for processing on the request queue; before processing is complete, the application needs more DOPs for additional operations. If the application can allocate all the additional DOPs it requires, the application might consume all available system memory. However, when you specify a high number of DOPs to the Get Free DOPs, the application halts further memory allocation until that number of DOPs is available for reuse.

Video Device Driver Introduction

Return Queue Characteristics

Because DOPs are allocated in two sizes, it is logical to regard the return queue as actually two queues, one where small DOPs are returned and another where large DOPs are returned. When you issue the Get Free DOPs QIO, you specify the number of DOPs 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 can use an alternate return queue to share return queues on a per-process or systemwide basis. To do this, specify the address of a return queue structure as the fourth parameter of the Get Viewport ID QIO when you create the viewport. (In this case, the return queue section of the viewport DOP queue structure is ignored.) See Appendix B for a description of this structure.

1.3.12 Deferred Queue

When a portion of a viewport is occluded onscreen, one option is to write to an update region in offscreen memory. Sometimes, however, offscreen memory is so crowded that you cannot keep an update region there. In that case, you must save the writing operations on the *deferred queue*.

The driver stores all operations directed to a portion of an unavailable viewport to the QDSS hardware on the deferred queue. Deferred queue operations can be executed whenever the previously occluded portion of the viewport becomes available.

NOTE: 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 informs an application the deferred queue is full.

The Execute Deferred Queue QIO executes the operations on the deferred queue. Chapter 2 contains additional information on deferred queue operations.

- 2 An optional user-defined AST parameter 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 where 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 that indicates which button is activated; the transition values are 400, 401, 402, and 403. The system assigns these values 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 and 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 pressed. You can use these keys in combination as *meta-keys* as follows:

- Bit 14 corresponds to the Shift key
- Bit 13 corresponds to the Control key
- Bit 12 corresponds to the Lock key
- The address of a pointer button characteristics block that determines whether delivery of subsequent transitions depends on all buttons being in the up position. By default, the specified button transition AST gets every transition 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 associated with the button transition AST.

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

- Place the entry last on the list (QV_LAST)
- Delete the entry (QV_DELETE)
- Purge the type-ahead 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 with the current pointer position, it fires the associated button transition AST.

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

Programming to the Driver

Example 2-7 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 ; 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
:
:

```

Example 2-7 illustrates the typical programming and use of pointer button ASTs. (Appendix D shows this example in the context of a complete applications program.)

2.6 Using the Type-Ahead Buffer

From 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 processes it. When this happens, the character and button information is stored in the *type-ahead buffer*. (Pointer movement inputs the new pointer position, but if the input cannot be delivered, it is ignored, not buffered.)

The type-ahead buffer is part of each entry on the list. It is 128 bytes long, so you can buffer 32 input tokens (each token is four bytes long).

2.6.1 Getting Input from the Type-Ahead Buffer

You can obtain input from the type-ahead buffer in two ways:

- When you enable the entry, associate a repeating AST with it to process buffered input continuously. (For keyboards, you can also associate a repeating AST when you modify the keyboard.)

2

Programming to the Driver

This chapter describes the following:

- How to perform programming tasks when you write to the video device drivers
- How to perform tasks common to both drivers or specific to either the QVSS or the QDSS driver
- How to use specific QIOs and DOPs in combination to perform tasks

Chapter 3 and Chapter 4 explain each function of the QIO interface in detail. Chapter 5 describes the QDSS system DOPs available for drawing to the display.

2.1 Initializing the Screen

You must initialize the screen before you can write to it. Initialization places the VAXstation screen in a known state. Once initialization is complete, an application can issue QIOs and begin screen operations.

NOTE: You must initialize the screen before you perform a draw operation. If you fail to do this, the drawing operation will not work properly.

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

To obtain 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.

Using the QDB

If a QDSS application is using only the UISDC DOP interface, it does not have to access the QDB. However, under some circumstances (for example, if it manipulates the pointer position or requires tablet information), it might need the system information stored in the QDB. Example 2-12, later in this chapter, shows how an application uses the QDB to get the systemwide viewport ID.

Programming to the Driver

Using the QVB

Typically, before a QVSS application can perform a drawing operation, it must obtain specific video memory information from the QVB:

- Starting address of video memory to set bits (draw) in memory
- Address of the scanline map to map to the screen any lines in memory it wants to display

Section 2.10 describes how to use the QVB for drawing operations. 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$_QV_GETSYS),
2          QDB_DESC,,,)           ! address of descri
IF (STATUS .NE. 1) THEN
    CALL LIB$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$SL_MAIN_VIDEOADDR
SCAN = QVB.QVB$SL_MAIN_MAPADDR

RETURN
END
```

2.3 Using Channels with Video Device Drivers

The drivers use channel numbers to identify application list entries. The drivers track various events by listing event entries (see Section 1.3.4). When an application uses the QIO interface to place an entry on a list, 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. (See *VAX/VMS System Services Reference Manual*.)

For each application list entry, the channel number must be unique to that list. That is, if a process uses channel 1 to create an entry on the keyboard list, it cannot use channel 1 to create another entry on the keyboard list. It must obtain another channel to create another entry. When it adds or deletes entries, the driver uses the channel number to identify each entry uniquely.

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

2.4 Using the Keyboard

Driver keyboard functions enable an application to perform the following operations:

- Receive keyboard input
- Modify keyboard characteristics
- 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 with the Enable Keyboard Input QIO. This creates an entry on the keyboard entry list. When you enable a keyboard, you can specify:

- The address of a four-longword AST specification block. The four longwords contain the following information:
 - 1 The address of an AST routine that determines what action to take when input is received. This is called a keystroke AST because it is fired for each keystroke entered.

If no AST routine is specified, input is stored in the type-ahead buffer and delivered either when an AST region is declared with 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).

Programming to the Driver

4 A zero.

- The address of an AST routine invoked whenever the entry is made active (brought to the top of the list). This is called a request AST (or control AST). It is fired only once for each entry activation. Only one keyboard can be active at a time. Typically, the request AST performs any necessary actions that the enabled keyboard requires; for example, it pops an obscured window to make displayed input visible.
- The address of a keyboard characteristics block that describes the characteristics of the keyboard. Each keyboard is associated with a set of characteristics (key click, auto repeat, and so on) defined in this data structure. You can choose 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 can determine the position of the entry in the list by specifying optional modifiers to the QIO. The application can use the modifiers to perform the following operations:

- Cycle the list, which moves the top entry to the end of the list (QV_CYCLE)
- Place the entry last on the list (QV_LAST)
- Delete the entry (QV_DELETE)
- Purge the type-ahead buffer (QV_PURG_TAH)

Typically, an application defines one keyboard for each window. However, it can define the characteristics of each keyboard differently from window to window. This feature permits you to create “virtual” keyboards. Although there is only one physical keyboard, you can define a number of different keyboards, and an application can enable any number of keyboards as long as it keeps track of them. When you no longer need a keyboard, delete it as follows:

- Use the QV_DELETE modifier with the QIO, or
- Deassign the associated channel

Example 2-1 illustrates:

- A typical assignment of two terminal channels
- Keyboard requests on those channels
- Associated AST routines

(Appendix D shows this example in the context of a complete application program.)

Example 2-1 Enabling Keyboard Requests

```

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

P3_BLOCK:
    .LONG   CTL_AST          ; control AST specification
    .LONG   0                ; block - for both ASTs
    .LONG   0                ; 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
.
.
.
KBD_AST:
F5_AST:
    .WORD    4(AP)           ; send acknowledgment
    PUSHL   4(AP)
    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

```

Example 2-1 Cont'd. on next page

Programming to the Driver

Example 2-1 (Cont.) Enabling Keyboard Requests

```
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

Keyboards have a set of default characteristics associated with them. These default characteristics are defined by a data structure called the system characteristics block. Appendix A illustrates this data structure and lists the default keyboard characteristics.

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 with the following parameters:

- Longword 1—Bit mask of the characteristics to enable
- Longword 2—Bit mask of the characteristics to disable
- Longword 3—Key-click volume value in the range 1 to 8 (1 is loudest).
- Longword 4—Screen saver time, in minutes

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

Once you modify the systemwide defaults, if you enable a keyboard without specifying characteristics, the keyboard assumes the new default values.

To define the keyboard characteristics (auto repeat, key-click sound, function key transition) for a particular keyboard entry, specify the address of a keyboard characteristics block as the fourth parameter of the Enable Keyboard Input QIO. This block also consists of four longwords with the following parameters:

- Longword 1—Bit mask of the characteristics to enable
- Longword 2—Bit mask of the characteristics to disable
- Longword 3—Key-click volume value in the range 1 to 8 (1 is loudest).

Note that Longword 4 must be zero.

You can enable the same characteristics for a specific keyboard as for the systemwide defaults, except for the screen saver time, which is a systemwide characteristic (there is only one screen). See Appendix A.

For example, assume an application enables two keyboards, one with autorepeat and the other without autorepeat. When one keyboard is active, holding down any key causes it to be entered repeatedly; when the other keyboard is active, the key is entered only once.

To modify the characteristics of an existing keyboard, specify a keyboard characteristics block as the fourth parameter of the Modify Keyboard Characteristics QIO. Note that you can also use this QIO 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 can associate the keys of the keyboard with any of the 255 characters in the multinational character set (including diacritical characters). (Appendix I shows the multinational character set.) An application can define several character sets to be accessed at different times by the same physical keyboard.

To define a character set, construct a data structure (keyboard table), then use the Load Keyboard Table QIO to load the new table.

2.4.3.1

Constructing a Keyboard Table with Macros

To construct a keyboard table, initialize it with the default table values, then override any values you want to modify. `SY$LIBRARY:$QVBDEF.MLB` contains the following macros to generate keyboard tables:

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

These macros are described in Appendix E.

Initializing a Table

Call `VC$KEYINIT` to initialize a table. This macro has one parameter, the address of the table, which it *returns* after allocating space and initializing the table. Specify this parameter and use the returned address when you load the table. By default, the system loads the North American keyboard table.

Loading Key Definitions

Call `VC$KEY` to load new key definitions. Several parameters permit you to define the various states of a given key. Note that you modify only the keys that are different from the default keys. Loading key definitions *overrides* the default definitions loaded by `VC$KEYINIT`.

Depending on how you press the Shift, Control, and Lock keys in combination with it, a keyboard key can have eight different states. For each key, the keyboard table associates each state with a one-byte ASCII value that represents a character from the multinational character set. Each

Programming to the Driver

key is described by a quadword in the table. Table 2-1 lists the key states and the byte within the quadword that describes each state.

Table 2-1 Key States

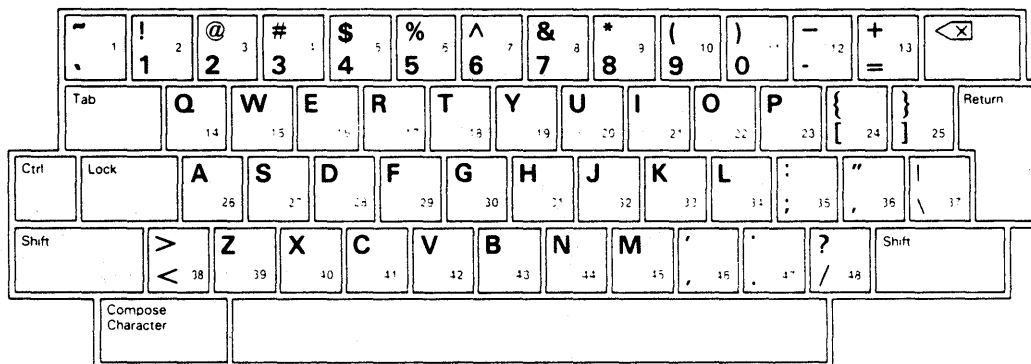
Byte	State
1	Value of key
2	Value of key if Shift key is also pressed
3	Value of key if Control key is also pressed
4	Value of key if both Shift and Control keys are also pressed
5	Value of key if Lock is also pressed
6	Value of key if both Lock and Shift keys are also pressed
7	Value of key if both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

When you use VC\$KEY to load a key, you specify the following information:

- Nine parameters
- Ordinal key position
- 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 small print). This table corresponds to the North American keyboard character layout.

Figure 2-1 Keyboard Table Layout



ZK-4450-85

The following example shows the loading of 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 denotes an undefined key (no character is delivered).

Table 2-2 shows the decimal values that represent diacritical characters. (Section 2.4.5 contains additional information on diacritical characters.)

Table 2-2 Diacritical Characters

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

Terminating a Table

Call VC\$KEYEND to terminate a keyboard table. This macro *returns* one parameter, the length of the table. Specify this parameter and use the returned length when you load the table.

Example 2-2 shows how you can modify the North American keyboard layout. (Appendix D shows this example in the context of a complete application program.)

2.4.3.2 Constructing a Keyboard Table Without Macros

It is possible to construct a keyboard table without using the provided macros. Such a keyboard table must conform to the structure illustrated in Figure 2-2.

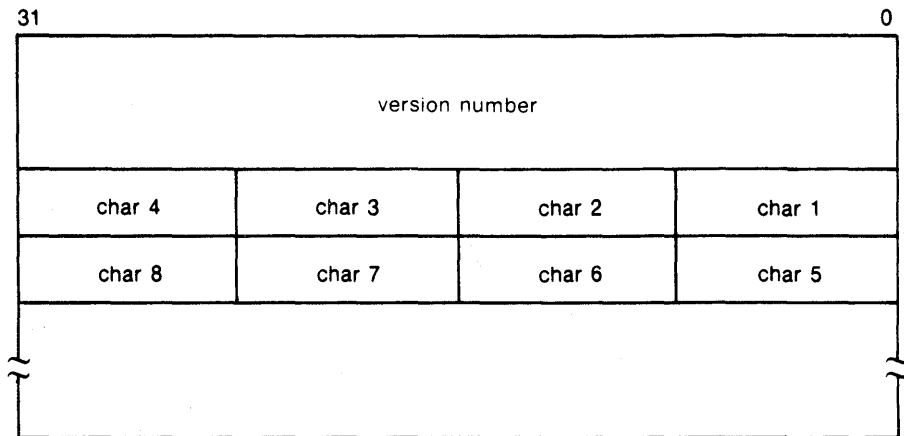
Example 2-2 Modifying the North American Keyboard

```

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

```

Figure 2-2 Keyboard Table Description



ZK 5347 86

- The first quadword of the table must contain the table version number. Typically this value is 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 associate the table. Note that the VC\$KEYINIT and VC\$KEYEND macros return address and length, respectively.

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

You can revert to the default keyboard table by calling the Revert to Default Keyboard Table QIO.

NOTE: If a private table was loaded, this QIO also returns the space used to pool.

Example 2-3 is a typical routine for loading a keyboard table. (Appendix D shows this example in the context of a complete application program.)

Example 2-3 Loading 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
.
.
.

```

2.4.4 Composing Nonstandard Characters

Compose Sequences

Use *compose sequences* to define combinations of keys to represent multinational characters not already defined as standard keys in the keyboard table.

Depending on your keyboard, you can use two types of compose sequences:

- Three-stroke sequences—Press the Compose key, then press two standard keys. All keyboards support three-stroke sequences.
- Two-stroke sequences—Press a *diacritical* mark, then press a standard key. The North American keyboard does not support two-stroke sequences.

Programming to the Driver

Diacritical Marks

A diacritical mark is one of the following nonspacing characters:

Grave accent—È
Acute accent—É
Circumflex accent—Ê
Tilde—Ñ
Diaeresis (umlaut)—Ë
Ring—Å

Diacritical marks are available on all but the North American keyboard. (This is why you cannot perform two-stroke sequences on the North American keyboard.) Diacritical marks vary among the keyboards according to the relative usage of characters with diacritical marks. Note that only one of several characters shown on a key cap can be a diacritical mark; some keyboards have keys with both a standard character and a diacritical mark.

To define compose sequences, construct a *compose sequence table* data structure (either two-stroke, three-stroke, or both), then load the table with the Load Compose Sequence Table QIO.

2.4.5 Constructing Compose Sequence Tables

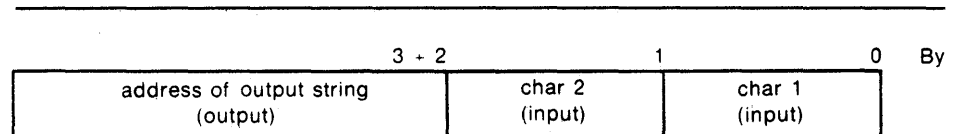
A compose sequence table lists a series of compose sequences. The structures of three-stroke and two-stroke tables differ slightly.

Three-Stroke Compose Sequence Table Structure

Three-stroke compose tables have three parts:

- 1 A longword with the version number for the table (typically this value is 1).
- 2 A series of longwords that list the two standard keys used in the compose sequence and hold an address that points to the associated output string, in the format shown in Figure 2-3.

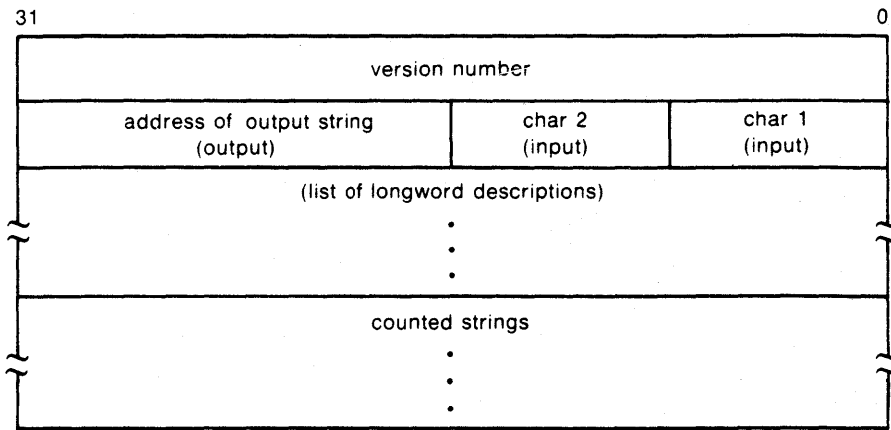
Figure 2-3 Three-Stroke Compose Sequence Table Description



- 3 Counted string that bytes 2 and 3 of the longword (address of output string) point to. All 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



ZK-4452-85

Two-Stroke Compose Sequence Table Structure

Two-stroke compose sequence tables have four parts:

- 1 A longword with the table version number (typically this value is 1).
- 2 A 32-byte diacritical table that 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. Thus, you can define nonstandard diacritical characters. For example, if you set bit 65 (decimal), the uppercase letter "A" is a diacritical character. If you set bit 112 (decimal), the lowercase letter "p" is a diacritical character.

To support *standard* diacritical characters, represent the characters with the decimal values shown in Table 2-3.

Table 2-3 Diacritical Characters

Diacritical Mark	Equivalent Character	Decimal Value
Diaeresis (umlaut)	Ä	128
Acute accent	Á	129
Grave accent	À	130
Circumflex accent	Â	131
Tilde	Ñ	132
Ring	Ö	133
(Reserved)		134-159

- 3 A series of longwords that list the diacritical key and the standard key used in the compose sequence and hold an address that points to the associated output string, in the format shown in Figure 2-5.

2.4.5.1 Using Macros to Construct Compose Sequence Tables

To construct a compose table, initialize it, then load the sequences you want to define. VMS Workstation Software provides macros to generate compose tables in SYS\$LIBRARY:\$VWSSYSDEF.MLB:

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

These macros are also described in Appendix F.

Initializing a Table

Call VC\$COMPOSE_KEYINIT to initialize a table. This macro has two parameters:

- The address of the table, which it *returns* after it allocates space and initializes the table. Specify this parameter and use the returned address when you load 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

Call VC\$COMPOSE_KEY to load a compose sequence. 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. Example 2-4 illustrates loading a three-stroke compose sequence.

Example 2-4 Loading a Three-Stroke Compose Sequence

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

Terminating a Table

Call VC\$COMPOSE_KEYEND to terminate a keyboard table. This macro *returns* one parameter, the length of the table. Typically, you specify this parameter and use the returned length when you load the table.

Programming to the Driver

2.4.5.2 Loading a Compose Table

To load a compose table, use the Load Compose Sequence Table QIO with the address and size of the table and the channel of the keyboard entry with which you want to associate the table. The VC\$COMPOSE_KEYINIT and VC\$COMPOSE_KEYEND macros return address and length, respectively.

The VMS Workstation is shipped with copies of the Digital standard three-stroke and two-stroke compose tables that reside in the driver. These tables are the default until you load alternates.

NOTE: Digital standard two-stroke compose sequences are not supported on the North American keyboard.

To revert to the default compose table, call the Revert to Default Compose Table QIO.

Example 2-5 illustrates how to load a three-stroke compose table. (Appendix D shows this example in the context of a complete application program.)

Example 2-5 How to Load a Three-Stroke Compose 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
.
.
.
```

2.5 Using a Pointer Device

The drivers detect two pointer-related conditions:

- Pointer movement
- Pointer button transition

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

The driver uses separate lists to track 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

Use the Enable Pointer Movement QIO to create a pointer movement entry. When you create a pointer movement entry, specify the following information:

- The address of a four-longword AST specification block. The four longwords contain the following parameters:
 - 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 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 where the driver stores the new cursor position so it is accessible to the application.

The low-order word of the longword holds the 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 the Y pixel position and ranges from 0 to 863, where 0 is the bottom of the screen.

- The address of an AST routine invoked whenever the pointer *exits* the specified region. Typically, this AST performs any necessary cleanup actions. For example, it turns off a region highlighted by the action AST.
- The 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 can determine the position of the entry in the list by specifying optional modifiers to the QIO. The modifiers can be used to perform the following functions:

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

Programming to the Driver

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 specify an exit AST, the driver fires that AST when it discovers that the pointer position is no longer within the specified region.

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

Example 2-6 illustrates how to program a pointer motion AST. (Appendix D shows this example in the context of a complete applications program.)

Example 2-6 How to Program a Pointer Motion AST

```
.
.
.
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
      .LONG  MOUSE_AST                ; specification block
      .LONG  MOUSE_ACK                ; AST address
      .LONG  0                        ; AST parameter
      .LONG  0                        ; access mode
      .LONG  MOUSE_XY                 ; new pointer cursor position
      .LONG  0                        ; 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

Use the Enable Button Transition QIO to create a pointer button transition. When you create a button transition entry, specify the following information:

- The address of a four-longword AST specification block. The four longwords contain the following parameters:
 - 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) is stored in the type-ahead buffer and delivered either when you declare an AST region with the same channel or when you issue a Get Next Input Token QIO with the same channel.

- 2 An optional user-defined AST parameter 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 where 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 that indicates which button is activated; the transition values are 400, 401, 402, and 403. The system assigns these values 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 and 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 pressed. You can use these keys in combination as *meta-keys* as follows:

- Bit 14 corresponds to the Shift key
- Bit 13 corresponds to the Control key
- Bit 12 corresponds to the Lock key
- The address of a pointer button characteristics block that determines whether delivery of subsequent transitions depends on all buttons being in the up position. By default, the specified button transition AST gets every transition 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 associated with the button transition AST.

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

- Place the entry last on the list (QV_LAST)
- Delete the entry (QV_DELETE)
- Purge the type-ahead 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 with the current pointer position, it fires the associated button transition AST.

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

Example 2-7 illustrates the typical programming and use of pointer button ASTs. (Appendix D shows this example in the context of a complete applications program.)

Programming to the Driver

Example 2-7 Typical Programming and Use of Pointer Button ASTs

```
.
.
.
SET_BUTTONAST:
    $ASSIGN_S    DEVNAM=WS_DEVNAM,- ; assign channel using
                CHAN=BUT_CHAN      ; logical name and
                ; channel number
                ; no error if set
    BLBS        R0,10$              ; error
    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
                ; no error if set
    BLBS        R0,20$
    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 Type-Ahead Buffer

From 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 processes it. When this happens, the character and button information is stored in the *type-ahead buffer*. (Pointer movement inputs the new pointer position, but if the input cannot be delivered, it is ignored, not buffered.)

The type-ahead buffer is part of each entry on the list. It is 128 bytes long, so you can buffer 32 input tokens (each token is four bytes long).

2.6.1 Getting Input from the Type-ahead Buffer

You can obtain input from the type-ahead buffer in two ways:

- When you enable the entry, associate a repeating AST with it to process buffered input continuously. (For keyboards, you can 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 can have an AST associated with it—in either case the input is delivered in the IOSB block). This type of single-token processing is called a “one shot.”

- Issue a Get Next Input Token QIO to process a single input token from the buffer (this QIO can have an AST associated with it—in either case, the input is delivered in the IOSB). This type of single-token processing is called a “one shot.”

Once a repeating AST is associated with an entry, attempts to issue subsequent one-shot ASTs on that entry return an error because the results are unpredictable. If you enable an entry without an associated AST, you can issue one-shot ASTs 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-shot ASTs are processed first.

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

2.7 Intercepting Input

You can issue one-shot ASTs 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), then issue one shots. (Note that for a keyboard you can disable an AST by modifying the keyboard instead of reenabling it.) Later, you can reenable the repeating AST.

To intercept the input for an entry on a list, use the Get Next Input Token QIO; specify the type of input token (IO\$C_QV_ENAKB, IO\$C_QV_MOUSEMOV, or IO\$C_ENABUTTON) and the channel with which the entry is associated.

You can also use one shot ASTs 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 AST with no associated AST delivers the input to the IOSB. With an event flag and the IOSB, the application can process the type-ahead 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 enables you to associate a region of the screen with a specific cursor pattern. Use the QIO interface to change the shape and size of the cursor to reflect a change in functionality; for example, an editing cursor can take one shape while a menu selection cursor takes another. Again, the driver maintains an entry list to keep track of cursor patterns.

Use the Define Pointer Cursor Pattern QIO to create a cursor pattern entry. When you create a cursor pattern entry, specify the following information:

- The address of a bitmap image for the new cursor pattern. This bitmap image is a 16-word array on single-plane cursor systems or a 32-word array on multiplane cursor systems. The QVB contains a field that indicates whether you have a single-plane or multiplane system; use

Programming to the Driver

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 enables 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 can determine the position of the entry in the list by specifying optional modifiers to the QIO. The modifiers can be used to perform the following functions:

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

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

Example 2-8 illustrates the typical assignment of a single-plane cursor region pattern. (Appendix D of this manual shows this example in the context of a complete application program.)

2.8.1 Multiplane Cursor Patterns

If your system uses a multiplane cursor, you can specify a 32-word array as a cursor pattern. Multiplane cursors consist of two planes. Typically, you use two planes to prevent the cursor from disappearing when it moves 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 as follows:

- 1—Indicates that the corresponding pixel be filled.
- 0—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 as follows:

Programming to the Driver

- 1—Indicates that the corresponding pixel be filled with the background color.
- 0—Indicates that the corresponding pixel be filled with the foreground color.

2.9 Using an Alternate Windowing System

For 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, before you invoke the STARTVWS.COM command procedure, modify the command procedure SYSTARTUP_V5.COM to define the logical name UIS\$WS_ALTAPPL to "TRUE."

This instructs the driver to reserve half of video memory for a private application. At the request of the private application, this part of video memory is mapped to the screen and becomes 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 dynamically between windowing systems.

NOTE: Private applications are device dependent; only one private application can be active at a time.

2.10 Drawing to the QVSS Screen

To draw to the screen using the QVSS driver, follow these 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 setting bits directly in video memory. To access video memory, use the QVB\$L_MAIN_VIDEOADDR address in the QVB. The application determines how to offset into video memory. When you manipulate memory, remember the following:

- Each scanline of video memory is 1024 bits (128 bytes) wide.
- There are 1024 scanlines in memory.

NOTE: If you use an alternate windowing system, 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, 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 indices of scanline positions in video memory. The index of scanlines in memory starts at zero and is 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. To ensure that you are mapping the correct portion of memory, calculate the correct scanline base in video memory. To obtain the correct scanline base, complete the following steps:

- 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 you insert it as an entry in the scanline map.

2.11 Creating a QDSS Viewport

The QDSS driver performs all viewport operations to the screen. If your application is not using the UIS windowing interface, it must create a viewport or use the systemwide viewport before it attempts to write to the screen. Example 2-12, later in this chapter, 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

Use the \$ASSIGN system routine to obtain a unique channel for a viewport. 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

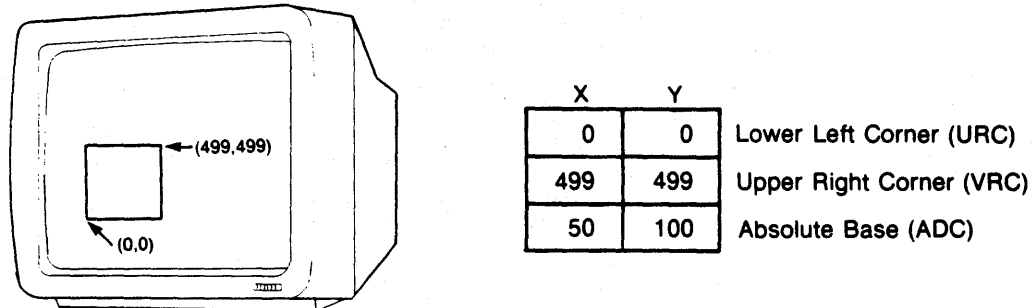
Use the Get Viewport ID QIO to get a viewport ID. One parameter of this QIO specifies the address of the longword to receive the ID. The ID stored at that address identifies which viewport is the object of all subsequent operations. The channel the application specifies in this QIO is associated with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

2.11.3 Defining a Viewport

A viewport is defined by one or more rectangular update regions. Update regions are defined in 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 can 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.

Programming to the Driver

Figure 2-7 Viewport and Update Region Definition Buffer



ZK-5348-86

Note that the base is given in absolute coordinates and the two defining corners of the viewport are given in viewport relative coordinates. These coordinates become 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, follow these steps:

- 1 Allocate and initialize one or more URD buffers that describe the viewport's size and relative position.
- 2 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 and length of the viewport definition buffer and the viewport ID.

To redefine a viewport, reinvoke 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 with the Start Request Queue QIO.

Example 2-9 creates and starts a 100-pixel square viewport with its lower left corner at the absolute device coordinate (10,10). Note that in FORTRAN, you must include the IODEF library to access the QIO function codes.

Example 2-9 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

! 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$_QD_GET_VIEWPORT_ID),
2             VIEWPORT_ID,              ! address of ID buffer
2             %VAL(4),                  ! VP ID buffer length
2             ' ',)
IF (STATUS .NE. 1) THEN
    CALL LIB$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$_QD_SET_VIEWPORT_REGIONS),
2             VP_URD,                    ! address of URD buffer
2             %VAL(URD$_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

```

Example 2-9 Cont'd. on next page

Programming to the Driver

Example 2-9 (Cont.) Creating a Viewport

```
! 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

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.

You must allocate storage for DOPs, insert DOPs on the request queue to execute them, and reuse the storage with the return queue. If your application uses the UIS windowing environment, you can perform all three of these functions with UISDC routines. However, if your application does not use the UIS windowing environment, the application must manage DOP storage and insert DOPs on the request queue.

Chapter 5 describes how to perform drawing with DOPs.

2.13 Using Bitmaps

Although the QDSS driver does not support direct manipulation of the onscreen bitmap, it permits you to copy bitmap images from processor memory to onscreen and offscreen memory and from onscreen 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 Chapter 4 for details. The UISDC interface also provides a Load Bitmap function. (See Chapter 5.)

To draw an image, complete the following steps:

- 1 Build the image in processor memory.
- 2 Use the Write Bitmap QIO to load the image into QDSS memory.

To store an image in processor memory, use the Read Bitmap QIO to copy the bitmap from QDSS memory to processor memory. (Complete this process for occluded viewport regions when offscreen memory is full; see Section 2.18.)

Use the Load Bitmap QIO to load a bitmap for use with a DOP.

Example 2-10 later in this chapter illustrates the use of the Write Bitmap QIO to copy a region from onscreen memory into offscreen memory (bitmap-to-bitmap transfer).

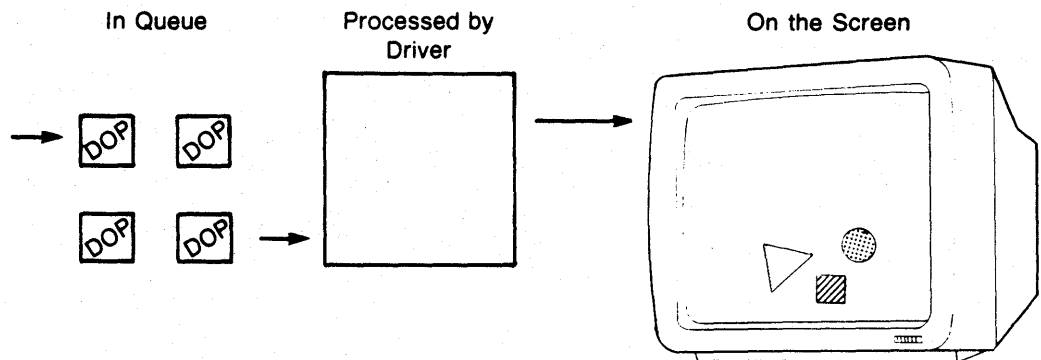
2.14 Synchronizing Viewport Activity

Because DOPs are queued asynchronously for processing and execution, you must take special action to synchronize activity on a viewport.

Figure 2-8 illustrates the three DOP states:

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

Figure 2-8 Synchronizing Viewport Activity



ZK-5349-86

The driver can process a number of DOPs at a time. To synchronize activity, manipulate the queue and be aware of whether the driver is currently processing DOPs.

Programming to the Driver

Synchronization QIOs

The QIO interface provides the following QIOs 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 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 performs a synchronous DOP).

Request Queue Interface DOPs

In addition to the 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.

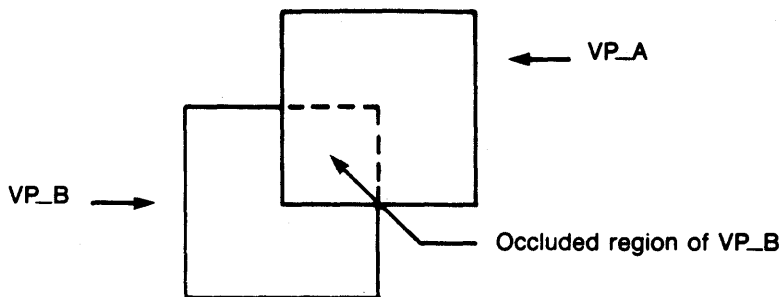
A Stop also differs from a Suspend in that a Stop issued on a stopped Request Queue waits for the queue to restart, then takes effect, while a Suspend issued on a suspended viewport is ignored. The Stop is thus useful for synchronizing multiprocess windowing activity on a single viewport. To guarantee that no other process accesses the viewport, a process can issue a Stop before it attempts any windowing activity (redefining URDs and so forth). When control returns from the Stop, it is clear 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, two or more viewports might overlap. This overlapping is called occlusion. Figure 2-9 illustrates one viewport (VP_A) occluding another (VP_B).

Figure 2-9 Occluded Viewport



ZK-5350-86

Only one viewport can display the overlapped area of the bitmap at a time. If your application permits occlusion, it must be able to handle any operations directed to an occluded viewport region. It does this by moving the occluded region of the viewport into offscreen memory and performing any necessary operations there. If that portion of the screen becomes available for display later, you can *pop* the viewport, or copy the up-to-date region back to the screen. 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 originally defined as one rectangle with a single URD can be redefined as a number of rectangles (viewport regions) with one 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 one another.

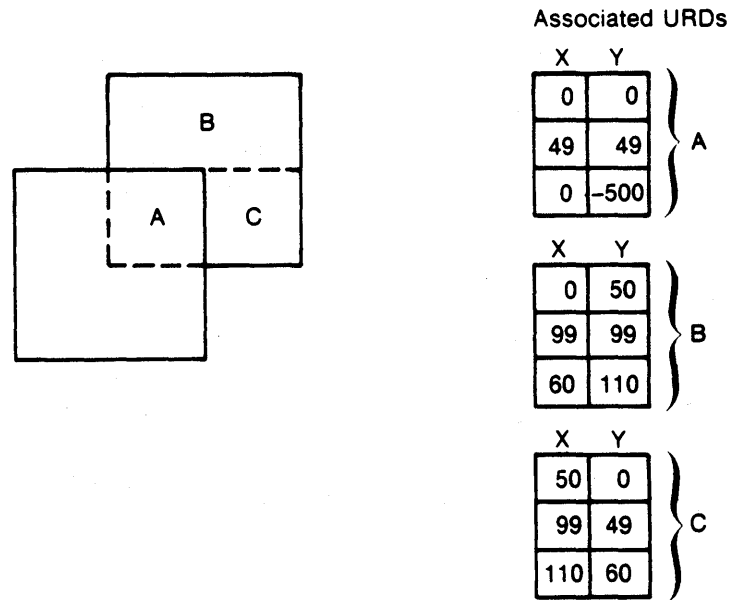
Use the Define Viewport Region QIO to redefine a viewport. You can use a negative Y coordinate to redefine an occluded region in offscreen memory; because drawing operations use viewport-relative coordinates, the drawing is performed properly. You must ensure that the negative Y coordinate you use falls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

Figure 2-10 illustrates 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.

Programming to the Driver

The base coordinates of each accessible rectangle are in absolute device coordinates 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.

Figure 2-10 Redefining Viewports with URDs



ZK-5351-86

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 simultaneously to the same piece of the display. Before it creates a viewport, your application should determine whether another viewport already exists in the area of the screen where the viewport will be created. (The application is responsible for tracking each viewport on the screen.)

To secure exclusive access to a viewport bitmap follow these steps:

- 1 Use the Stop Request Queue QIO to stop activity on the existing viewport to ensure a known state for the subsequent steps.
- 2 Use the Define Viewport Region QIO to redefine the regions of the existing viewport.
- 3 Use the Write Bitmap or Read Bitmap QIO to copy the to-be-occluded region of the existing viewport to offscreen memory.

- 4 Update the URD definition of the existing viewport to reflect its new state. Specify a negative Y coordinate in the base value of the URD to redefine the occluded region to be offscreen. (The negative Y value must fall within the range of the free area of offscreen memory shown in Figure 1-4). Drawing can still be performed to offscreen memory.
- 5 Use the Start Request Queue QIO to restart the viewport.

Now you can create the new viewport on screen and start drawing operations on it.

Example 2-10 illustrates how one viewport occludes 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 by the predefined structure in the VWSSYSDEF file.

Programming to the Driver

Example 2-10 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$_QD_STOP),        ! QD function code
2           %VAL(VP1_ID),             ! address of ID buffer
2           '...')
IF (STATUS .NE. 1) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Load TPB for occluded rectangle
CALL LOAD_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)
```

Example 2-10 Cont'd. on next page

Example 2-10 (Cont.) Securing Bitmap Access

```

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

! 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

```

Example 2-10 Cont'd. on next page

Programming to the Driver

Example 2-10 (Cont.) Securing Bitmap Access

```
! 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)

CODE = IO$_SETMODE
STATUS = SYS$QIOW (,
2          %VAL(VP_CHANNEL),          ! channel
2          %VAL(CODE),                ! QIO function code
2          ' ',
2          %VAL(IO$_QD_SET_VIEWPORT_REGIONS),
2          URD,                        ! address of URD buffer
2          %VAL(3 * URD$_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
```

2.15.3 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 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 URDs.
- 8 Restart the (formerly) occluded viewport.

Example 2-11 illustrates popping a viewport.

Example 2-11 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
.
.
.

! Stop VP2 (the occluding viewport)
CODE = IO$_SETMODE
STATUS = SYSS$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$_QDWRITE
STATUS = SYSS$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)

```

Example 2-11 Cont'd. on next page

Programming to the Driver

Example 2-11 (Cont.) Popping a Viewport

```
! Restart VP2
CODE = IO$ SETMODE
STATUS = SYSSQIOW (,
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

! Stop VP1 (the occluded viewport)
CODE = IO$ SETMODE
STATUS = SYSSQIOW (,
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 = SYSSQIOW (,
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 = SYSSQIOW (,
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

.
.
.

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

Example 2-11 Cont'd. on next page

Example 2-11 (Cont.) Popping a Viewport

```

SUBROUTINE LOAD_TP2 (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_TP3 (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$W_X_SOURCE = 0                  ! x of lower left corner
TPB.TPB$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

```

Example 2-11 Cont'd. on next page

Programming to the Driver

Example 2-11 (Cont.) Popping a Viewport

```
! 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)

! 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$_OD_SET_VIEWPORT_REGIONS),
2          URD,                        ! address of URD buffer
2          %VAL(URD_LENGTH),          ! length of URD buffer
2          %VAL(VIEWPORT_ID),,,)      ! address of VP ID
IF (STATUS .NE. 1) THEN
CALL LIB$SIGNAL (%VAL (STATUS))
END IF

RETURN
END
```

2.16 Deleting a Viewport

When you delete 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 drawing is completed, you can deassign the associated channel to ensure that nothing else is written to the viewport. Finally, you can erase the viewport. The following sections describe each procedure.

2.16.1 Synchronizing Viewport Deletion

Before you deassign a channel, you must ensure that all drawing to a viewport is complete, as follows:

- 1 Issue a Stop Viewport Activity DOP with the Insert DOP QIO (the QDWRITE function code with the IO\$M_QD_INSERT_DOP modifier) to stop pending operations either on the DOP request queue or in progress before the delete. This QIO waits for the stop to occur before returning control, which accounts for the lag time in processing DOPs. If you do not wait for completion, you might delete the viewport while DOPs are on the queue.
- 2 Use the \$DASSGN system service to deassign the associated channel.

2.16.2 Erasing a Viewport

To erase a viewport, use the Fill Polygon DOP to draw a background-colored rectangle over the viewport. The channel of the viewport to be erased is already disassociated. You must use the systemwide viewport to perform this operation as follows:

- 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 background-colored rectangle over the viewport (see Chapter 5 for details about DOPs). You must have the systemwide viewport ID to perform this DOP on the systemwide viewport.

Example 2-12 illustrates deleting a viewport.

Programming to the Driver

Example 2-12 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

! 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
```

Example 2-12 Cont'd. on next page

Example 2-12 (Cont.) Deleting a Viewport

```

! 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          ' ',
2          %VAL(IO$_QV_GETSYS),
2          QDB_DESC, ' ',          ! address of descriptor
IF (STATUS .NE. 1) THEN
    CALL LIB$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$_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$_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
.
.
.
! *****
! 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

```

Example 2-12 Cont'd. on next page

Programming to the Driver

Example 2-12 (Cont.) Deleting a Viewport

```
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

.
.
.

! *****
! * 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_SYSPV

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

! 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'
```

Example 2-12 Cont'd. on next page

Example 2-12 (Cont.) Deleting a Viewport

```

! 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

```

.17 Moving a Viewport

The QDSS driver does support moving a viewport or changing its size. However, an application can move a viewport as follows:

- 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.

.18 Using the Deferred Queue

An application is responsible for tracking offscreen memory use. When a viewport is occluded and the free area of offscreen memory is already full of occluded regions, you can ensure a drawing operation for the region only by placing the region on the deferred queue as follows:

- 1 To save the state of the region until update, use the Read Bitmap QIO to copy the region to processor memory.

Programming to the Driver

- 2 Use the Define Viewport Region QIO to redefine the region, setting the absolute base coordinates to (-1,-1). When you place a region on the deferred queue, the relative coordinates are used only to inform the driver that operations for the region are to be stored on the deferred queue.

Before you use the deferred queue, call the Notify Deferred Queue Full QIO. This QIO enables 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 it is notified that the queue is full. To execute an operation on the deferred queue, follow these steps:

- 1 Use the Write Bitmap QIO to copy the region back into offscreen memory.
- 2 Use the Define Viewport Region QIO to redefine the region.
- 3 Use the Execute Deferred Queue QIO to execute operations stored on the queue.

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

Note that if a viewport is occluded in more than one place, you might 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 so, define a region on the deferred queue when you redefine the first region in offscreen memory for deferred queue execution. This step informs the driver that the viewport still has an occluded region and prevents it from deleting the deferred queue.

After the deferred queue drawing operations are executed to all the deferred regions of a viewport, use the Delete Deferred Queue Operation QIO to delete the drawing operations from the deferred queue. 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 map to a single pixel on the display. The number of system-configured memory planes 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 can display 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 can display 256.

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

Each displayable color is represented by a value in the *hardware color map* (the hardware look-up table). On color systems, a color is represented by one 16-bit intensity value 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 represent the actual intensity values, which 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 follows:

- 0—Color system
- 1—Intensity system

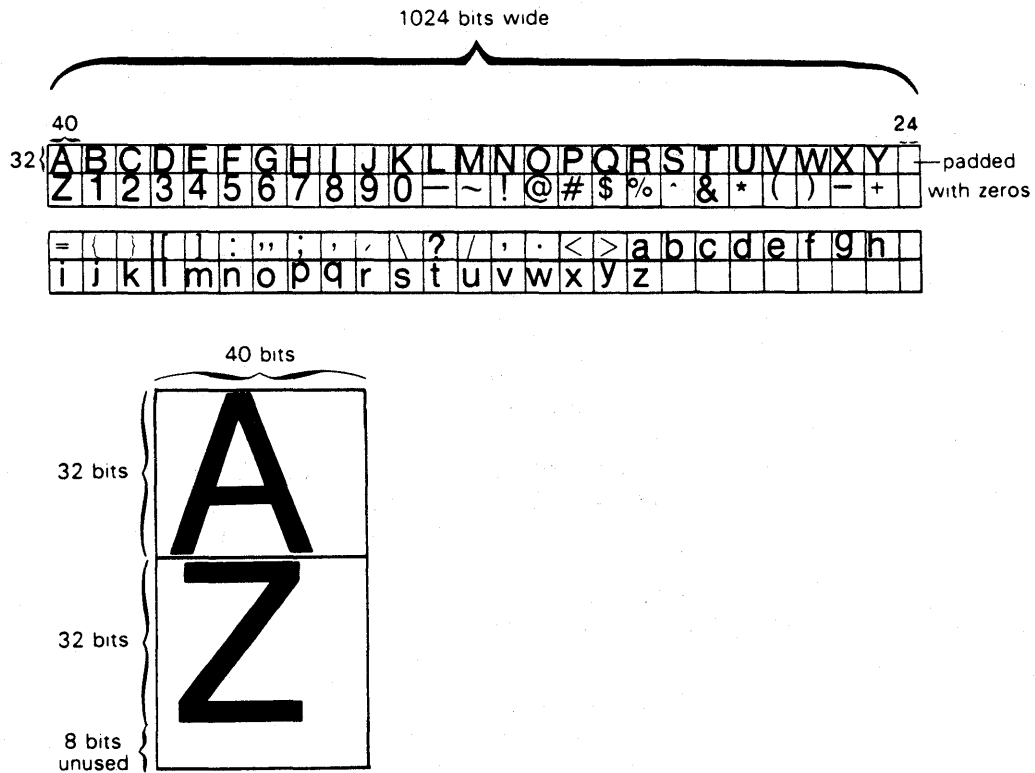
Once the identification is complete, the driver accepts only Set Color Map Entries QIO requests that match this setting. The driver rejects all other Set Color Map Entries QIOs.

2.19.2 Manipulating Color Map Values

The values that 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 this.

Programming to the Driver

Figure 2-11 Indexing the Hardware Color Map



ZK 5477 86

If your application is not using the UIS environment, it must load any values it uses into the hardware color map with the Set Color Map Entries QIO, and specify the following information:

- Index into the color map at which to begin initialization
- Address of the buffer that contains the desired intensity values
- Length of the buffer

Call this QIO at any time to redefine the values in the color map.

To determine the current values of the hardware color map, the application should use the Get Color Map Entries QIO and specify the following information:

- Index into the color map at which to begin information retrieval
- Address of the buffer that holds the returned intensity values
- Length of the buffer

3

QVSS/QDSS Common QIO Interface

This chapter contains an alphabetical listing of descriptions of the QIO calls you can use with the QVSS and QDSS drivers. Table 3-1 organizes the QIOs in functional groups.

Table 3-1 QIO 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 Movement }
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 you call QIOs, become familiar with Chapters 1 and 2 and Appendices A, B, and H.

- Chapters 1 and 2 describe the general operation of QIOs.
- Appendices A and B contain pictures and descriptions of the data types you pass to the driver through the P1 to P6 parameters.
- Appendix H describes the SYS\$QIO system service.

As you call QIOs, refer to the descriptions in this chapter.

3.1.1 QIO Description Format

The QIO descriptions follow a strict format. The main headings in each QIO description and the type of information that appears there follow:

QIO Name—Name of the QIO.

Overview—Brief description of the operation the QIO performs.

Format—Format of the call you must pass to SYS\$QIO to perform the desired operation.

Arguments in brackets are optional. Some programming languages, such as MACRO, allow you to omit optional arguments; 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 unspecified argument. Check the programming language documentation to see how the language handles optional arguments.

Unique Parameters—Information that passes to the driver through \$QIO P1 to P6 parameters; also indicates whether the parameter is required or optional.

Description—Additional QIO operation information.

Example—QIO example.

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 enable you to select the cursor style and reposition the pointer cursor.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 [,p2] [,p3] [,p4] [,p5] [,p6]*

UNIQUE PARAMETERS

P1 — IO\$_QV_SETCURSOR (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "OR" the IO\$_QV_SETCURSOR function code with one of the following optional function modifiers:

Function Modifier	Action
IO\$_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\$_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\$_QV_LAST	Places the specified pointer cursor pattern request last in the entry list. If IO\$_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.
IO\$_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\$_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\$_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.

QVSS/QDSS Common QIO Interface

Define Pointer Cursor Pattern

P2—Bitmap Image address (optional)

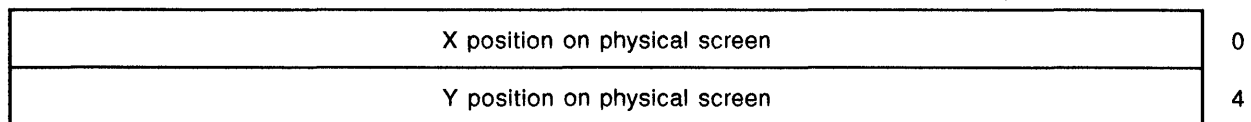
This parameter is either a 16-word array or, on multiplane cursor systems, a 32-word array. The QVB contains a field that informs you whether yours is a single-plane or multiplane system. Use the Get System Information QIO to access this field. (See Description section.)

P3—New cursor position address

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

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

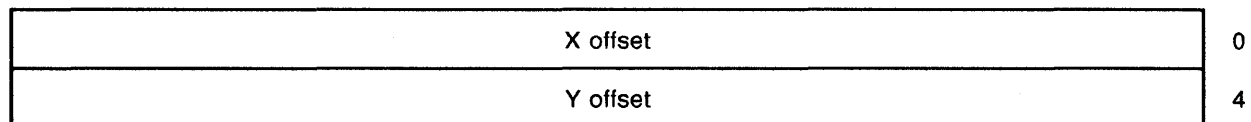
The following diagram shows the data structure that defines the new cursor position.



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

P4—Pointer cursor hot spot definition address

This parameter is an address that points to a two-longword array that defines the pointer cursor hot spot, the 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.

QVSS/QDSS Common QIO Interface Define Pointer Cursor Pattern

P5—Cursor style definition value

This parameter is a longword value in the range 0 through 3 that defines how the cursor is presented against the background screen.

This parameter is ignored for multiplane cursor systems.

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. 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. 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.

P6—Screen rectangle values block address (optional)

This parameter is a longword that points to a screen rectangle values block that defines a rectangle on the screen. If you do not specify P6, a default rectangle that covers the entire screen is used.

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

MINX (left side value)	0
MINY (bottom side value)	4
MAXX (right side value)	8
MAXY (top side value)	12

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

DESCRIPTION

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

QVSS/QDSS Common QIO Interface

Define Pointer Cursor Pattern

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 that is described in the following section). The shape can be in the form of a block, a cross, an arrow, or any other 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 can 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 as follows:

- 1—Indicates that the corresponding pixel must be filled.
- 0—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 as follows:

- 1—Indicates that the corresponding pixel must be filled with the background color.
- 0—Indicates that the corresponding pixel must be filled with the foreground color.

EXAMPLE

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

•
•
•

QVSS/QDSS Common QIO Interface

Enable Button Transition

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

UNIQUE PARAMETERS

P1—IO\$_QV_ENABUTTON (required)

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

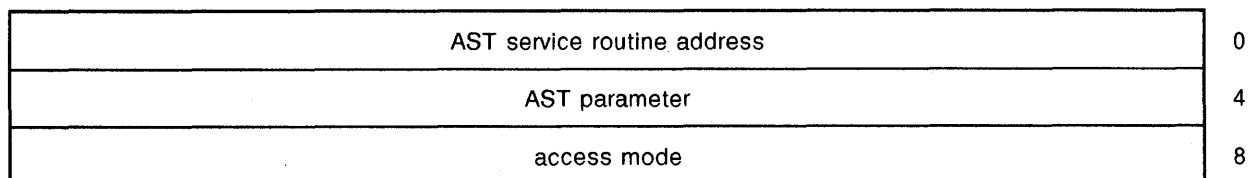
To modify the QIO action, "OR" the IO\$_QV_ENABUTTON function code with one of the following optional function modifiers:

Function Modifier	Action
IO\$_QV_DELETE	Deletes the specified pointer button request. Any data in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.
IO\$_QV_LAST	Places the specified pointer button request last in the list. If IO\$_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\$_QV_PURG_TAH	Purges the type-ahead buffer of any existing pointer button transitions.

P2—Pointer button AST specification block address (required)

This parameter is a longword that points to a pointer AST specification block that 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.



QVSS/QDSS Common QIO Interface Enable Button Transition

input token address

12

Field	Use
AST service routine address	AST service routine address 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 where the AST is delivered is maximized with the current access mode.
Input token address	<p>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 that indicates 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.</p> <p>Bit 15 of the high-order word determines whether the transition is up (0) or down (1). The remainder of the high-order word contains control information you can use to determine if the [Shift] (bit 12), [Ctrl] (bit 13), or [Lock] (bit 14) keys are pressed. You can use these as meta-keys (keys used in combination). When the bit is set, the key is down.</p>

P3—Must be 0

P4—Pointer button characteristics block address (optional)

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.

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

QVSS/QDSS Common QIO Interface

Enable Button Transition

enabled characteristics mask	0
disabled characteristics mask	4
0	8
0	12

Field	Use				
Enabled characteristics mask	Longword of characteristics to be enabled.				
Disabled characteristics mask	Longword of characteristics to be disabled. The pointer button characteristics, defined by the \$QVBDEF macro, consist of the following bit:				
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Characteristic</th> <th style="text-align: left;">Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_BUT_UPTODOWN</td> <td>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.</td> </tr> </tbody> </table>	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.
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—Screen rectangle values block address (optional)

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.

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

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

QVSS/QDSS Common QIO Interface Enable Button Transition

MINX (left side value)	0
MINY (bottom side value)	4
MAXX (right side value)	8
MAXY (top side value)	12

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

DESCRIPTION

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

QVSS/QDSS Common QIO Interface

Enable Button Transition

EXAMPLE

The following example shows typical programming and use of pointer button ASTs.

```
SET_BUTTONAST:
    $ASSIGN_S   DEVNAM=WS_DEVNAM,- ; ASSIGN CHANNEL USING
                CHAN=BUT_CHAN    ; LOGICAL NAME AND
                                ; CHANNEL NUMBER
    BLBS       RO,10$             ; NO ERROR IF SET
    BRW        ERROR             ; ERROR
10$:          MOVL   #IO$C_QV_ENABUTTON,RO
    $QIOW_S    CHAN=BUT_CHAN,-
                FUNC=#IO$SETMODE,-
                P1=(RO),-
                P2=#BUT_BLOCK,-
                P6=#BUT_REGION
    BLBS       RO,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 enables you to use the tablet as a data digitizer.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 ,p3 [,p4] ,p5 [,p6]*

UNIQUE PARAMETERS

P1—IO\$_C_QV_ENABLE_DIGITIZING (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "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 in the type-ahead buffer is delivered to the specified AST address before the delete operation is executed.

P2—Pointer movement AST specification block address (optional)

This parameter is a longword that points to a pointer movement AST specification block that specifies a user-supplied AST routine. The routine is notified when pointer movement occurs inside the data rectangle specified 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	0
AST parameter	4
access mode	8
address of new pointer cursor position	12

QVSS/QDSS Common QIO Interface

Enable Data Digitizing

Field	Use
AST service routine address	The AST service routine address 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 where the AST is delivered is maximized with the current access mode.
New pointer cursor position address	The fourth longword contains the address of a longword that receives 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; range of X is defined in the <i>qvb\$w_tablet_width</i> field of the QVB block. The high-order word receives the new Y pixel location of the cursor; range of Y is defined in the <i>qvb\$w_tablet_height</i> field of the QVB block.

P3—Pointer button AST specification block address (optional)

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	0
AST parameter	4
access mode	8
input token address	12

QVSS/QDSS Common QIO Interface Enable Data Digitizing

Field	Use
AST service routine address	The AST service routine address 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 where the AST is delivered is maximized with the current access mode.
Input token address	This 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 (0) or down (1). The rest of the high-order word contains more control information that can be used to determine if the Shift (bit 14), Ctrl (bit 13), or Lock (bit 12) keys are pressed. You can use these keys as meta-keys (keys used in combination). When the bit is set, the key is down.

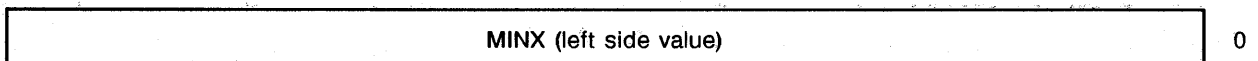
P4, P5—Must be 0

P6—Data rectangle values block address (optional)

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

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

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



QVSS/QDSS Common QIO Interface

Enable Data Digitizing

MINY (bottom side value)	4
MAXX (right side value)	8
MAXY (top side value)	12

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

DESCRIPTION

Only the process that issues the data digitizing request can change or cancel it. When the process is deleted, any outstanding data digitizing is canceled.

Only one data digitizing region can 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

Enables 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 **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 ,p3 [,p4] [,p5] [,p6]*

**UNIQUE
PARAMETERS**

P1—IO\$C_QV_ENAFNKEY (required)
This function code identifies the action the QIO performs.

P2—Reserved function keystroke AST specification block address (required)

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	0
AST parameter	4
access mode	8
input token address	12

Field	Use
AST service routine address	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.

QVSS/QDSS Common QIO Interface

Enable Function Keys

Field	Use
Access mode	This is the access mode where the AST is delivered. It is maximized with the current access mode.
Input token address	This 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 signaled only on a down transition.

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 value 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

```

QVSS/QDSS Common QIO Interface

Enable Function Keys

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

UNIQUE PARAMETERS

P1—IO\$_C_QV_SIMULATE (required)

This function code identifies what action the QIO performs. If you set the TYPE field in the string descriptor you use to TYPE_T2 (value 38), the string is evaluated as 16-bit characters rather than 8-bit characters, and any 16-bit value can be passed as the low word for keyboard input.

NOTE: The LENGTH field in the descriptor is the number of 16-bit characters rather than a byte-count.

P2—ASCII text descriptor address (optional)

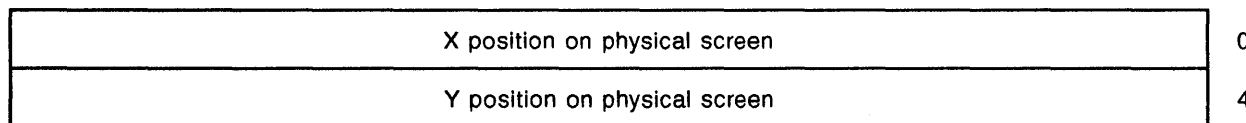
This parameter is a longword that points to a descriptor for the ASCII text to send to the current keyboard region. The maximum number of characters allowed in the text string is 32. If P2 is 0, no data is sent.

P3—New pointer position address (optional)

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

If P3 is 0, the pointer is not repositioned.

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



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

P4—Button simulation block address (optional)

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

If P4 is 0, the pointer buttons are not modified.

The following diagram shows a button simulation block.

QVSS/QDSS Common QIO Interface Enable Input Simulation

buttons to be pressed mask	0
buttons to be released mask	4
0	8
0	12

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										
	Pointer button definitions used in the masks defined by the \$QVBDEF macro, consisting of the following symbols:										
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Symbol</th> <th style="text-align: left;">Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_BUTTON_1</td> <td>Select button</td> </tr> <tr> <td>QV\$M_BUTTON_2</td> <td>Button 2</td> </tr> <tr> <td>QV\$M_BUTTON_3</td> <td>Button 3</td> </tr> <tr> <td>QV\$M_BUTTON_4</td> <td>Button 4</td> </tr> </tbody> </table>	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
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.										

P5, P6—Must be 0

QVSS/QDSS Common QIO Interface

Enable Input Simulation

EXAMPLE

The following example 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 **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 ,[p3] ,[p4] [,p5] [,p6]*

UNIQUE PARAMETERS

P1—IO\$_QV_ENAKB (required)

This function code identifies the action the QIO performs.

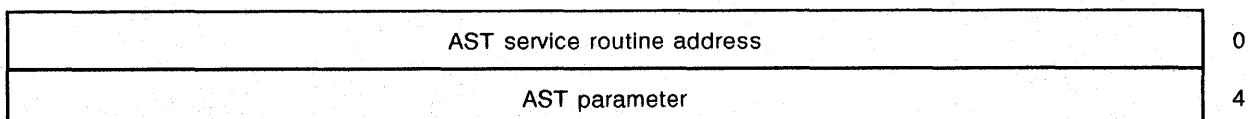
To modify the QIO action, "OR" the IO\$_QV_ENAKB function code with one of the following optional function modifiers:

Function Modifier	Action
IO\$_QV_CYCLE	Removes the active keyboard from the top 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\$_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\$_QV_LAST	Places the specified keyboard request last in the list. If IO\$_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\$_QV_PURG_TAH	Purges the type-ahead buffer of any keyboard request on this channel.

P2—Keystroke AST specification block address (required)

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.



QVSS/QDSS Common QIO Interface

Enable Keyboard Input

access mode	8
input token address	12

Field	Use
AST service routine address	The AST service routine address 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 where the AST is delivered is maximized with the current access mode.
Input token address	<p>This value 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 from 0 to 255 map into the Digital multinational character set. Values from 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.</p> <p>The rest of the high-order word contains more control information that can be used to determine if the Shift (bit 12), Ctrl (bit 13), or Lock (bit 14) keys are pressed. You can use these keys as meta-keys (keys used in combination). When the bit is set, the key is down.</p>

P3—Keyboard request AST specification block address (optional)

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.

QVSS/QDSS Common QIO Interface Enable Keyboard Input

AST service routine address	0
AST parameter	4
access mode	8
0	12

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 where the AST is delivered is maximized with the current access mode.
0	The fourth longword must be zero.

P4—Keyboard characteristics block address (optional)

This parameter is a longword address that points to a keyboard characteristics block that describes keyboard-related characteristics 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 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	0
disabled characteristics mask	4
keyclick volume	8
0	12

QVSS/QDSS Common QIO Interface

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, defined by the \$QVBDEF macro, consist of the following bits:	
Characteristic	Default	Meaning
QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.
QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.
QV\$M_KEY_UDF6	Off	Function keys F6 through F10 generate up/down transitions.
QV\$M_KEY_UDF11	Off	Function keys F11 through F14 generate up/down transitions.
QV\$M_KEY_UDF17	Off	Function keys F17 through F20 generate up/down transitions.
QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.
QV\$M_KEY_UDE1	Off	Function keys E1 through E6 generate up/down transitions.
QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.
QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.
Keyclick volume	The keyclick volume is a value from 1 (loudest) to 8 (softest). If a value of 0 is specified, the current system default keyclick volume is used.	
0	The fourth longword must be 0.	

P5, P6—Must be 0

EXAMPLE

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

QVSS/QDSS Common QIO Interface

Enable Keyboard Input

```

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
    .LONG   CTL_AST        ; BLOCK
    .LONG   0              ; 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

```

QVSS/QDSS Common QIO Interface

Enable Keyboard Input

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

.
.
.

Enable Keyboard Sound

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

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 [,p3] [,p4] [,p5] [,p6]*

**UNIQUE
PARAMETERS**

P1—IO\$_C_QV_SOUND (required)

This function code identifies the action the QIO performs.

P2—Symbolic name that denotes type of sound (required)

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

Characteristic	Meaning
QV\$_SOUND_BELL	Sound bell.
QV\$_SOUND_CLICK	Sound keyclick.

P3—Value that specifies the sound volume (optional)

This parameter specifies the sound volume, a value from 1 (loudest) to 8 (softest). If a value of 0 is indicated, the previously specified (that is, current) volume is used.

P4, P5, P6—Must be 0

QVSS/QDSS Common QIO Interface

Enable Keyboard Sound

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

UNIQUE PARAMETERS

P1—IO\$_QV_MOUSEMOV (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "OR" the IO\$_QV_MOUSEMOV function code with one of the following optional function modifiers:

Function Modifier	Action
IO\$_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\$_QV_LAST	Places the specified pointer motion request last in the list. If IO\$_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—Pointer motion AST specification block address (required)

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.

AST service routine address	0
AST parameter	4
access mode	8
address of new pointer cursor position	12

QVSS/QDSS Common QIO Interface

Enable Pointer Movement

Field	Use
AST service routine address	The AST service routine address 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 where the AST is delivered is maximized with the current access mode.
New pointer cursor position address	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 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 from 0 through 1023, with the lowest value denoting the left side of the screen; Y is from 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 <i>qvb\$w_tablet_width</i> field of the QVSS block; the range of Y is defined in the <i>qvb\$w_tablet_height</i> field of the QVB block.

P3—Pointer cursor exit AST specification block address (optional)

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 shows the data structure that specifies a pointer cursor exit AST.

AST service routine address	0
AST parameter	4
access mode	8
0	12

QVSS/QDSS Common QIO Interface Enable Pointer Movement

Field	Use
AST service routine address	The AST service routine address 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 where the AST is delivered is maximized with the current access mode.
0	The fourth longword must be zero.

P4, P5—Must be 0

P6—Screen rectangle values block address (optional)

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

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

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

MINX (left side value)	0
MINY (bottom side value)	4
MAXX (right side value)	8
MAXY (top side value)	12

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

DESCRIPTION

The QVSS and QDSS drivers track the pointer by moving the *pointer cursor* on the physical screen. 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.

QVSS/QDSS Common QIO Interface

Enable Pointer Movement

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
      .LONG   MOUSE_AST                 ; SPECIFICATION BLOCK
      .LONG   MOUSE_ACK                 ; AST ADDRESS
      .LONG   0                         ; AST PARAMETER
      .LONG   0                         ; ACCESS MODE
      .LONG   MOUSE_XY                  ; NEW MOUSE CURSOR POSITION
      .LONG   0                         ; 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 at the top of the list receives a control AST when a cycle request occurs. A cycle request is an entry control AST request that includes the IO\$M_QV_CYCLE function modifier.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 [,p2] ,p3 [,p4] [,p5] [,p6]*

UNIQUE PARAMETERS

P1—IO\$C_QV_ENAUSER (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "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 beginning 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 on the list. If an outstanding entry control request exists for the channel, it is updated to reflect the new priority.

P2—Must be 0 (required)

P3—Active entry AST specification block address (required)

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.

QVSS/QDSS Common QIO Interface

Enable User Entry

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

AST service routine address	0
AST parameter	4
access mode	8
0	12

Field	Use
AST service routine address	The AST service routine address 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 where the AST is delivered is maximized with the current access mode.
0	The fourth longword must be 0.

P4, P5, P6—Must be 0

QVSS/QDSS Common QIO Interface

Get Keyboard Characteristics

Field	Use																														
Disabled characteristics mask	<p>The second longword is a mask of characteristics that are disabled.</p> <p>The keyboard characteristics, defined by the \$QVBDEF macro, consist of the following bits:</p> <table border="1"> <thead> <tr> <th>Characteristic</th> <th>Default</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_KEY_AUTORPT</td> <td>On</td> <td>Key held down automatically repeats.</td> </tr> <tr> <td>QV\$M_KEY_KEYCLICK</td> <td>On</td> <td>Keyclick sounds on each keystroke.</td> </tr> <tr> <td>QV\$M_KEY_UDF6</td> <td>Off</td> <td>Function keys F6 through F10 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF11</td> <td>Off</td> <td>Function keys F11 through F14 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF17</td> <td>Off</td> <td>Function keys F17 through F20 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDHELPDO</td> <td>Off</td> <td>Function keys HELP and DO generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDE1</td> <td>Off</td> <td>Function keys E1 through E6 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDARROW</td> <td>Off</td> <td>Arrow keys generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDNUMKEY</td> <td>Off</td> <td>Numeric keypad keys generate up/down transitions.</td> </tr> </tbody> </table>	Characteristic	Default	Meaning	QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.	QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.	QV\$M_KEY_UDF6	Off	Function keys F6 through F10 generate up/down transitions.	QV\$M_KEY_UDF11	Off	Function keys F11 through F14 generate up/down transitions.	QV\$M_KEY_UDF17	Off	Function keys F17 through F20 generate up/down transitions.	QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.	QV\$M_KEY_UDE1	Off	Function keys E1 through E6 generate up/down transitions.	QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.	QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.
Characteristic	Default	Meaning																													
QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.																													
QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.																													
QV\$M_KEY_UDF6	Off	Function keys F6 through F10 generate up/down transitions.																													
QV\$M_KEY_UDF11	Off	Function keys F11 through F14 generate up/down transitions.																													
QV\$M_KEY_UDF17	Off	Function keys F17 through F20 generate up/down transitions.																													
QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.																													
QV\$M_KEY_UDE1	Off	Function keys E1 through E6 generate up/down transitions.																													
QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.																													
QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.																													
Keyclick volume	<p>The keyclick volume must be between 1 (loudest) to 8 (softest). If you specify 0, the current system default keyclick volume is used.</p>																														
0	<p>The fourth longword must be 0.</p>																														

P5, P6—Must be 0

QVSS/QDSS Common QIO Interface

Initialize Screen

Initialize Screen

Initializes the screen to a known state.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 [,p2] [,p3] [,p4] [,p5] [,p6]*

UNIQUE PARAMETERS ***P1—IO\$_QV_INITIALIZE function code (required)***
This function code identifies the action the QIO performs.
P2, P3, P4, P5, P6—Must be 0

DESCRIPTION This QIO initializes the workstation screen. You must initialize the screen whenever you initialize a windowing system 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- and three-stroke compose sequence tables.

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

**UNIQUE
PARAMETERS**

P1—IO\$_QV_LOAD_COMPOSE_TABLE
(required)

This function code identifies the action the QIO performs.

To modify the QIO action, "OR" the IO\$_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 becomes the default table for the entire workstation. Specify this modifier only once after you initialize the workstation when you load the first table.

P2—Two-stroke compose sequence table size
(optional)

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

P3—Two-stroke compose sequence table address
(optional)

This parameter is a longword that points to the two-stroke compose sequence table. Chapter 2 describes the two-stroke compose sequence table.

P4—Three-stroke compose sequence table size
(optional)

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

P5—Three-stroke compose sequence table address
(optional)

This parameter is a longword that points to the three-stroke compose sequence table. Chapter 2 describes the three-stroke compose sequence table.

P6—Must be 0

QVSS/QDSS Common QIO Interface

Load Compose Sequence Table

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:

- Default table—Taken from the workstation default
- Private table—If one is loaded

EXAMPLE The following example shows how to load a three-stroke 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/>,<^0>
    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 **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 ,p3 [,p4] [,p5] [,p6]*

UNIQUE PARAMETERS

P1—IO\$_QV_LOAD_KEY_TABLE (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "OR" the IO\$_QV_LOAD_KEY_TABLE function code with the following optional function modifier:

Function Modifier	Action
IO\$_QV_LOAD_DEFAULT	If the IO\$_QV_LOAD_DEFAULT modifier is specified, this table becomes the workstation default. Specify this modifier only once when you load the first table after you initialize the workstation.

P2—Keyboard table size (required)

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

P3—Keyboard table address (required)

This parameter is a longword that points to the keyboard table. Chapter 2 describes keyboard tables.

P4, P5, P6—Must be 0

DESCRIPTION

Each window has two associated tables:

- Default table—Taken from the workstation default
- Private table—If one is loaded

QVSS/QDSS Common QIO Interface

Load Keyboard Table

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

**UNIQUE
PARAMETERS**

P1—IO\$_QV_MODIFYKB (required)

This function code identifies the action the QIO performs.

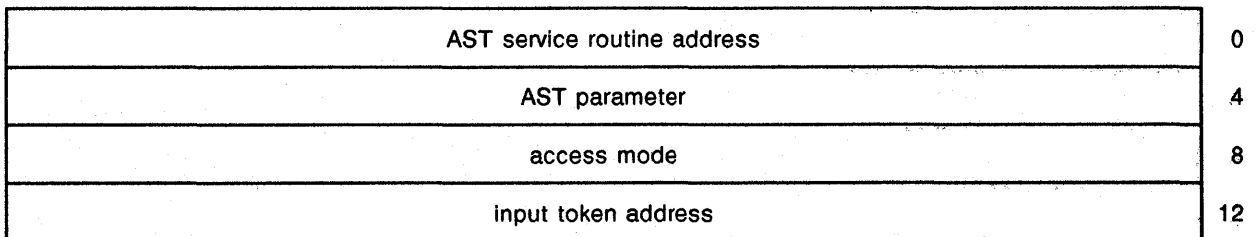
To modify the QIO action, "OR" the IO\$_QV_MODIFYKB function code with the following optional function modifier:

Function Modifier	Action
IO\$_QV_ACTIVE	Removes the active keyboard from the beginning 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—Keystroke AST specification block address (optional)

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.



QVSS/QDSS Common QIO Interface

Modify Keyboard Characteristics

Field	Use
AST service routine address	The AST service routine address 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 where the AST is delivered is maximized with the current access mode.
Input token address	This 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 from 0 to 255 map into the Digital multinational character set. Values from 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 signaled only on a down transition.

P3—Keyboard request AST specification block address (optional)

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.

AST service routine address	0
AST parameter	4
access mode	8
0	12

QVSS/QDSS Common QIO Interface Modify Keyboard Characteristics

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 where the AST is delivered is maximized with the current access mode.
0	The fourth longword must be zero.

P4—Keyboard characteristics block address (optional)

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

The default characteristics are those specified in the systemwide characteristics block, which you can modify with 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.

enabled characteristics mask	0
disabled characteristics mask	4
keyclick volume	8
0	12

Field	Use
Enabled characteristics mask	The first longword is a mask of characteristics to enable.

QVSS/QDSS Common QIO Interface

Modify Keyboard Characteristics

Field	Use																														
Disabled characteristics mask	<p>The second longword is a mask of characteristics to disable.</p> <p>The keyboard characteristics, defined by the \$QVBDEF macro, consist of the following bits:</p> <table border="1"> <thead> <tr> <th>Characteristic</th> <th>Default</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_KEY_AUTORPT</td> <td>On</td> <td>Key held down automatically repeats.</td> </tr> <tr> <td>QV\$M_KEY_KEYCLICK</td> <td>On</td> <td>Keyclick sounds on each keystroke.</td> </tr> <tr> <td>QV\$M_KEY_UDF6</td> <td>Off</td> <td>Function keys F6–F10 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF11</td> <td>Off</td> <td>Function keys F11–F14 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF17</td> <td>Off</td> <td>Function keys F17–F20 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDHELPDO</td> <td>Off</td> <td>Function keys HELP and DO generate up/down transitions..</td> </tr> <tr> <td>QV\$M_KEY_UDE1</td> <td>Off</td> <td>Function keys E1–E6 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDARROW</td> <td>Off</td> <td>Arrow keys generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDNUMKEY</td> <td>Off</td> <td>Numeric keypad keys generate up/down transitions.</td> </tr> </tbody> </table>	Characteristic	Default	Meaning	QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.	QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.	QV\$M_KEY_UDF6	Off	Function keys F6–F10 generate up/down transitions.	QV\$M_KEY_UDF11	Off	Function keys F11–F14 generate up/down transitions.	QV\$M_KEY_UDF17	Off	Function keys F17–F20 generate up/down transitions.	QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions..	QV\$M_KEY_UDE1	Off	Function keys E1–E6 generate up/down transitions.	QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.	QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.
Characteristic	Default	Meaning																													
QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.																													
QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.																													
QV\$M_KEY_UDF6	Off	Function keys F6–F10 generate up/down transitions.																													
QV\$M_KEY_UDF11	Off	Function keys F11–F14 generate up/down transitions.																													
QV\$M_KEY_UDF17	Off	Function keys F17–F20 generate up/down transitions.																													
QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions..																													
QV\$M_KEY_UDE1	Off	Function keys E1–E6 generate up/down transitions.																													
QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.																													
QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.																													
Keyclick volume	<p>The keyclick volume must be between 1 (loudest) and 8 (softest). If you specify 0, the current system default keyclick volume is used.</p>																														
0	<p>The fourth longword must be 0.</p>																														

P5, P6—Must be 0

Modify Systemwide Characteristics

Changes the systemwide windowing characteristics.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 [,p2] [,p3] [,p4] [,p5] [,p6]*

UNIQUE PARAMETERS **P1—IO\$_C_QV_MODIFYSYS (required)**
 This function code identifies the action the QIO performs.

P2, P3—Must be 0

P4—System characteristics block address (optional)

This parameter is a longword address that points to a system characteristics block, which specifies system-related characteristics to enable or disable.

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

enabled characteristics mask	0
disabled characteristics mask	4
keyclick volume	8
screen saver timeout value	12

Field	Use
Enabled characteristics mask	The first longword is a mask of characteristics to enable.

QVSS/QDSS Common QIO Interface

Modify Systemwide Characteristics

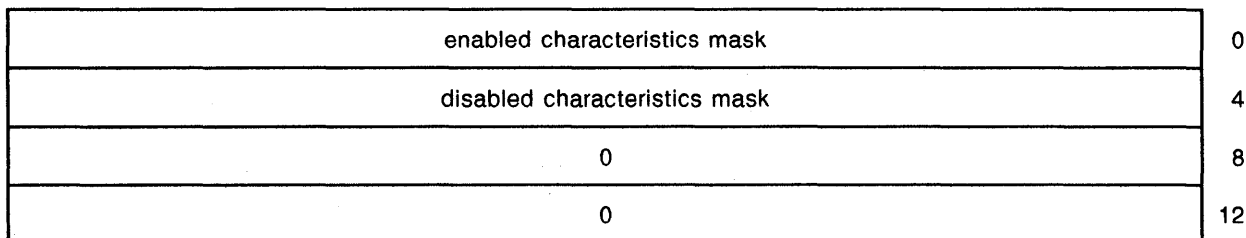
Field	Use																																	
Disabled characteristics mask	<p>The second longword is a mask of characteristics to disable.</p> <p>The system characteristics, defined by the \$QVBDEF macro, consist of the following bits:</p> <table border="1"> <thead> <tr> <th>Characteristic</th> <th>Default</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_KEY_AUTORPT</td> <td>On</td> <td>Key held down automatically repeats.</td> </tr> <tr> <td>QV\$M_KEY_KEYCLICK</td> <td>On</td> <td>Keyclick sounds on each keystroke.</td> </tr> <tr> <td>QV\$M_KEY_UDF6</td> <td>Off</td> <td>Function keys F6–F10 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF11</td> <td>Off</td> <td>Function keys F11–F14 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF17</td> <td>Off</td> <td>Function keys F17–F20 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDHELPDO</td> <td>Off</td> <td>Function keys HELP and DO generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDE1</td> <td>Off</td> <td>Function keys E1–E6 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDARROW</td> <td>Off</td> <td>Arrow keys generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDNUMKEY</td> <td>Off</td> <td>Numeric keypad keys generate up/down transitions.</td> </tr> <tr> <td>QV\$M_SYS_SCRSAV</td> <td>On</td> <td>Video output to monitor is disabled if no input activity occurs in the time specified in the fourth longword. Any keystroke, pointer button transition, or pointer motion resets the timer and reactivates a disabled screen.</td> </tr> </tbody> </table>	Characteristic	Default	Meaning	QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.	QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.	QV\$M_KEY_UDF6	Off	Function keys F6–F10 generate up/down transitions.	QV\$M_KEY_UDF11	Off	Function keys F11–F14 generate up/down transitions.	QV\$M_KEY_UDF17	Off	Function keys F17–F20 generate up/down transitions.	QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.	QV\$M_KEY_UDE1	Off	Function keys E1–E6 generate up/down transitions.	QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.	QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.	QV\$M_SYS_SCRSAV	On	Video output to monitor is disabled if no input activity occurs in the time specified in the fourth longword. Any keystroke, pointer button transition, or pointer motion resets the timer and reactivates a disabled screen.
Characteristic	Default	Meaning																																
QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.																																
QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.																																
QV\$M_KEY_UDF6	Off	Function keys F6–F10 generate up/down transitions.																																
QV\$M_KEY_UDF11	Off	Function keys F11–F14 generate up/down transitions.																																
QV\$M_KEY_UDF17	Off	Function keys F17–F20 generate up/down transitions.																																
QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.																																
QV\$M_KEY_UDE1	Off	Function keys E1–E6 generate up/down transitions.																																
QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.																																
QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.																																
QV\$M_SYS_SCRSAV	On	Video output to monitor is disabled if no input activity occurs in the time specified in the fourth longword. Any keystroke, pointer button transition, or pointer motion resets the timer and reactivates a disabled screen.																																
Keyclick volume	The keyclick volume must be a value from 1 (loudest) to 8 (softest). Default is 3.																																	
Screen saver timeout value	This value represents the number of minutes of inactivity that must elapse before the screen saver is activated. The value must be from 1 to 1440. If the value is 0, the timeout value stays unchanged. Default is 15.																																	

QVSS/QDSS Common QIO Interface Modify Systemwide Characteristics

P5—Pointer characteristics block address (optional)

This parameter is a longword address that points to a pointer characteristics block that specifies the pointer-related characteristics you 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, defined by the \$QVBDEF macro, consist of the following bits:						
	<table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; border-bottom: 1px solid black;">Characteristic</th> <th style="text-align: left; border-bottom: 1px solid black;">Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_PTR_ LEFT_HAND</td> <td>Inverts buttons on mouse or puck. (Buttons 1 and 3 are switched.)</td> </tr> <tr> <td>QV\$M_PTR_ INVERT_STYLUS</td> <td>Inverts buttons on stylus. (Buttons 1 and 3 are switched.)</td> </tr> </tbody> </table>	Characteristic	Meaning	QV\$M_PTR_ LEFT_HAND	Inverts buttons on mouse or puck. (Buttons 1 and 3 are switched.)	QV\$M_PTR_ INVERT_STYLUS	Inverts buttons on stylus. (Buttons 1 and 3 are switched.)
Characteristic	Meaning						
QV\$M_PTR_ LEFT_HAND	Inverts buttons on mouse or puck. (Buttons 1 and 3 are switched.)						
QV\$M_PTR_ INVERT_STYLUS	Inverts buttons on stylus. (Buttons 1 and 3 are switched.)						
0	Must be 0						
0	Must be 0						

P6—Must be 0

QVSS/QDSS Common QIO Interface

Modify Systemwide Characteristics

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:
. LONG     <QV$M_SYS_AUTORPT!QV$M_SYS_KEYCLICK> ; ENABLE
. LONG     <QV$M_SYS_UDF6!QV$M_SYS_UDARROW> ; DISABLE
. LONG     5 ; THESE CHARACTERISTICS
. LONG     30 ; THESE CHARACTERISTICS
. LONG     5 ; KEYCLICK VOLUME
. LONG     30 ; SCREEN SAVER TIMEOUT
.
.
.
```


QVSS/QDSS Common QIO Interface

Revert to Default Keyboard Table

Revert to Default Keyboard Table

Reverts to the default keyboard table.

FORMAT **SYSSQIO** *[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 (required)***

This function code identifies the action the QIO performs. The exclamation point (!) 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.

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.

4

QDSS-Specific QIO Interface

This chapter describes the QIOs you use only with the QDSS driver. The QIO descriptions appear in alphabetical order. Table 4-1 organizes the QIOs in functional groups.

Table 4-1 QDSS QIO Functional Groups

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 Operation } { 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 you call QIOs, read Chapters 1 and 2 and familiarize yourself with Appendices A, B, and H.

- Chapters 1 and 2 describe the general operation of QIOs.
- Appendices A and B contain diagrams and descriptions of the data types you pass to the driver through the P1 to P6 parameters.
- Appendix H describes the SYS\$QIO system service.

Then, as you call QIOs, refer to the descriptions in this chapter.

4.2 QIO Description Format

The QIO descriptions follow a strict format. The main headings in each QIO description and the type of information that appears there follow.

QIO Name—Name of the QIO.

Overview—Brief description of the operation the QIO performs.

Format—Format of the call you must pass to SYS\$QIO to perform the desired operation.

Arguments in brackets are optional. Some programming languages, such as MACRO, allow you to omit optional arguments; 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 unspecified argument. Check the programming language documentation to see how the language handles optional arguments.

Unique Parameters—Routines that pass information to the driver through \$QIO P1 to P6 parameters, which are noted as required or optional.

Description—Additional QIO operation information.

Example—QIO example.

Define Viewport Region

Creates or changes the update regions that compose a viewport.

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

UNIQUE PARAMETERS **P1—IO\$_QD_SET_VIEWPORT_REGIONS**
(required)

This function code identifies the action the QIO performs.

P2—Update region definition buffer address
(required)

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

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

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

P3—Update region definition buffer length
(required)

This parameter specifies the predefined constant URD\$_LENGTH multiplied by the number of update regions.

P4—Viewport ID (required)

This parameter is the 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.

P5, P6—Must be 0

QDSS-Specific QIO Interface

Define Viewport Region

DESCRIPTION

This function creates or modifies viewport update region definitions. The viewport that appears on the screen can consist of a number of update regions.

To access a viewport, you must call this function once. (A viewport must contain at least one update region.) If a viewport contains a number of regions, you make this call once, specifying a URD buffer (an array or record) that contains all the URDs.

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

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

QDSS-Specific QIO Interface

Execute Deferred Queue

Execute Deferred Queue

Processes any DOPs on the specified viewport deferred queue.

FORMAT **SYSSQIO** *[efn] ,chan , IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 [,p3] [,p4] [,p5] [,p6]*

UNIQUE PARAMETERS **P1—IO\$_C_QD_EXECUTE_DEFERRED (required)**
This function code identifies the action the QIO performs.

P2—Viewport_Id (required)
This parameter is the ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

P3, P4, P5, P6—Must be 0

DESCRIPTION This function 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 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 might occur.

Get Color Map Entries

Returns the values in the color map.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SENSEMODE ,[iosb]*
,[astadr] ,[astprm] ,p1 ,p2 ,p3 ,p4 [,p5] [,p6]

UNIQUE PARAMETERS

P1—IO\$_C_QD_GET_COLOR (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "OR" the IO\$_C_QD_GET_COLOR function code with one of the following optional function modifiers:

Function Modifier	Action
IO\$_M_QD_INTENSITY	Gets a map entry on an intensity system.
IO\$_M_QD_RESERVED_COLORS	Interprets the starting color map entry (P4) as the IO\$_C_QD_TWO_COLOR_CURSOR (set two-color cursor) parameter if the IO\$_M_QD_RESERVED_COLORS modifier is specified. The buffer here contains two map entries' worth of data: two words for intensity systems, six words for color systems.

P2—Color buffer address (required)

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

The size of the color buffer depends on the system:

- Color system—The buffer must have an "RGB triple" for each map entry you want to read. An "RGB triple" contains three word-long values—one for red, one for green, and one for blue. For example, for five color map entries, the color buffer must be 15 words long.
- Intensity system—The buffer must have a single word-long value for each map entry you want to read. For example, for five color map entries, the color buffer must be five words long.

Only the eight most significant bits are used for color definition.

P3—Length of the color buffer, in bytes (required)

The multiple you use depends on the system: color-6, intensity-2.

P4—Starting color map entry must be 0 (required)

This parameter is the color map index you use for the first RGB or intensity value specified in the buffer.

P5, P6—Must be 0

QDSS-Specific QIO Interface

Get Color Map Entries

DESCRIPTION This function enables an application to read the color map. Color map information is stored in the QDB, which you access by issuing a Get System Information QIO. Appendix B describes the QDB.

DESCRIPTION

This function returns the unique viewport identifier, which 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).

Call this function at viewport creation time after you have an assigned channel, before you call any function that requires a viewport ID. Call it only once for each channel; multiple calls result in multiple viewports.

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 **SYSS\$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 (required)
This longword points to the DOP entry for which you want completion notification.

P2—DOP entry length (required)
This longword contains the length of the DOP you are inserting into the queue.

P3—Viewport_id (required)
This parameter is the ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

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 enables 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 (rather than issuing a Stop Request Queue QIO), you guarantee that the DOPs inserted before the stop are executed.

Use this QIO only when your application requires notification that the request queue entry has finished executing. An INSQUE instruction or a UISDC\$QUEUE_DOP routine is much more efficient.

QDSS-Specific QIO Interface

Load Bitmap

Load Bitmap

Makes a bitmap available for use by a subsequent text, patterned line, move, or fill operation.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 ,p3 ,p4 ,p5 ,p6*

UNIQUE PARAMETERS

P1—IO\$_QD_LOAD_BITMAP (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "OR" the IO\$_QD_LOAD_BITMAP function code with the following optional function modifier:

Function Modifier	Action
IO\$_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 (required)

This longword points to the area of processor memory that describes the desired bitmap. You can also specify a zero in this parameter to postpone dynamic loading of the bitmap (see the Description section for details).

P3—Bitmap block length, in bytes (required)

This longword contains the length of the bitmap block in bytes. This value must be a multiple of 2.

P4—Address of a longword for the returned bitmap ID (required)

This longword is where 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.

P5—Bitmap width, in pixels (required)

This longword contains the width of the bitmap in pixels. The maximum bitmap width is 1024. If the bitmap is a single bit per pixel, you must specify a multiple of 16.

P6—Bits per pixel (required)

This longword contains the number of bits per pixel. Currently 1 and 8 are supported:

- 1—When a foreground and background color are sufficient (that is, for writing text)
- 8—When you want the full use of color (that is, for natural images)

DESCRIPTION

The Load Bitmap QIO makes a bitmap available to a drawing operation. Bitmaps specify the following features:

- Fonts
- Line styles
- Fill patterns
- Images

You load a bitmap from VAX memory to offscreen memory. Once the bitmap is in offscreen memory, drawing operations can access it with the bitmap ID. You 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 with the X,Y coordinates rather than a single index.

Bitmap storage is defined in blocks of video memory. The size of a bitmap block depends on whether you use a single-plane or multiplane image.

- Single-plane bitmap—Blocks are 70 bits by 1024 bits. Think of them as 70 lines, each 128 bytes wide.
- Multiplane bitmap—Blocks are 35 bits by 8180 bits. Think of them as 35 lines, each 1024 bytes long. Each consecutive eight bits in a line describes one pixel (in four-plane systems the high four bits are ignored).

If a bitmap requires more room than can be provided by a single bitmap block, store the bitmap in multiple blocks by getting a separate bitmap ID for each block and tracking them in your application.

Bitmaps are passed in blocks to the driver as follows:

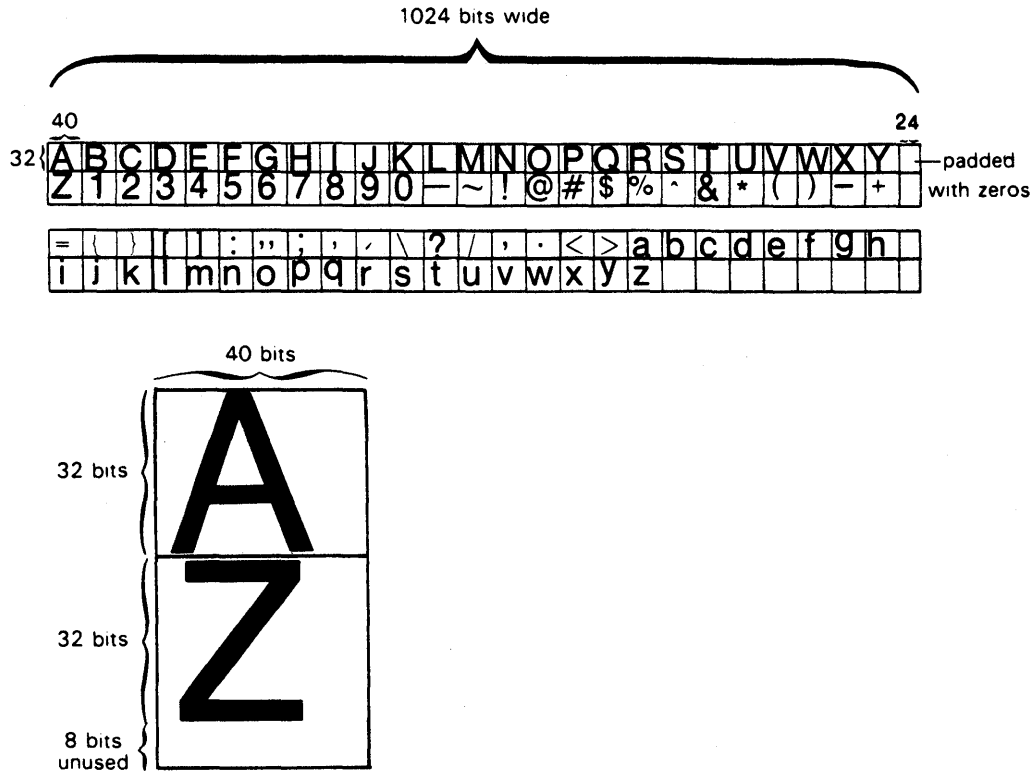
- Each bitmap character (glyph) is stored in a rectangle within the bitmap. Glyphs are stored sequentially and can be packed to the bit; however, it is more efficient to start each character on a byte boundary. (The nature of multiplane bitmaps is such that this occurs naturally.)
- If the total width of all glyphs exceeds the bitmap block limit, you must use another bitmap block, which you load separately.
- 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 shows 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 have to use another bitmap block.) Notice that the characters are aligned on word boundaries for better performance. If space is a greater consideration than performance, you can pack the characters in each bitmap block. Remember, your application must load each block separately and track the different bitmap IDs.

QDSS-Specific QIO Interface

Load Bitmap

Figure 4-1 Large Font Defined Across Bitmap Blocks



ZK-5477-86

Loading the Bitmap

Use Load Bitmap in two ways:

- 1 The bitmap is copied from the user buffer into a driver-maintained buffer. When the application accesses the bitmap, it is copied from the driver-maintained buffer into offscreen memory.
 - 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 must 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.
- 2 A handle (identifier) is created for the bitmap, but the application must supply the bitmap when it is accessed.
 - To load a bitmap dynamically, specify zero as the **bitmap address** parameter, but specify the correct length, width, and bits-per-pixel.

QDSS-Specific QIO Interface Load Bitmap

- 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).

The second method saves space because a bitmap is not loaded into offscreen memory until an application accesses it.

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.

Read Bitmap

Copies data from video memory to a user-specified buffer in VAX memory and performs bitmap-to-bitmap transfers.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_QDREAD ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 ,[p3] ,p4 ,p5 [,p6]*

**UNIQUE
PARAMETERS**

P1—Buffer address (required)

This parameter is the buffer address in VAX memory where 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.

P2—Buffer length

This longword contains the length in bytes of the buffer specified by P1.

P3—Must be 0

P4—Transfer parameter block (required)

The transfer parameter block (TPB) describes the bitmap to be read into VAX memory. The TPB contains:

- Coordinates for the lower left corner of the bitmap
- Bitmap height and width
- Predefined constant indicating the type of transfer
- For a bitmap-to-bitmap transfer, the coordinates for the lower left corner of the target bitmap

The following diagram shows the data structure that specifies the transfer parameters.

The following table lists the contents of each field in the Transfer Parameter block.

QDSS-Specific QIO Interface

Read Bitmap

Field	Use
TPB\$B_TYPE	Type of transfer being performed, either bitmap-to-processor (BTP) or bitmap-to-bitmap (BTB). Use the constants defined later to load this field.
TPB\$B_SIZE	Reserved to Digital
TPB\$W_X_SOURCE	X coordinate of lower left corner of source bitmap
TPB\$W_Y_SOURCE	Y coordinate of lower left 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 corner of target bitmap (only specified for bitmap-to-bitmap transfer)
TPB\$W_Y_TARGET	Y coordinate of lower left 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

The following table defines the constants in conjunction with the TPB.

Constant	Value
TPB\$C_BITMAP_XFR	Bitmap-to-bitmap transfer
TPB\$C_SOURCE_ONLY	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 (required)

This longword contains the length in bytes of the transfer parameter block specified in P4.

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 this function to perform bitmap-to-bitmap transfers by specifying a source and target location in the TPB and omitting the processor buffer.

QDSS-Specific QIO Interface

Resume Viewport Activity

Resume Viewport Activity

Resumes activity in a previously suspended viewport.

FORMAT	SYSSQIO <i>[efn] ,chan , IO\$_SETMODE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 [,p3] [,p4] [,p5] [,p6]</i>
---------------	--

UNIQUE PARAMETERS	<i>P1—IO\$C_QD_RESUME_VP (required)</i> This function code identifies the action the QIO performs.
--------------------------	--

	<i>P2—Viewport_id (required)</i> This parameter is the ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during viewport creation.
--	---

	<i>P3, P4, P5, P6—Must be 0</i>
--	--

DESCRIPTION	This function resumes activity on a viewport that was previously suspended with the Suspend Viewport Activity QIO function.
--------------------	---

QDSS-Specific QIO Interface

Set Color Map Entries

Set Color Map Entries

Defines (or alters) the color map.

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

UNIQUE PARAMETERS

P1—IO\$_QD_SET_COLOR (required)

This function code identifies the action the QIO performs.

To modify the QIO action, "OR" the IO\$_QD_SET_COLOR function code with one of the following optional function modifiers:

Function Modifier	Action
IO\$_QD_INTENSITY	A map entry is added on an intensity system.
IO\$_QD_RESERVED_COLORS	The starting color map entry (P4) is interpreted as the IO\$_QD_TWO_COLOR_CURSOR (set two-color cursor) parameter. The buffer must contain two map entries of data: two words for intensity systems, six words for color systems.

P2—Address of the color buffer (required)

P4—Must be 1

The color buffer differs depending on the system:

- Color system—The buffer must have an "RGB triple" for each map entry you set. An "RGB triple" contains three word-long values: one for red, one for green, and one for blue. For five color map entries, the color buffer must be 15 words long.
- Intensity system—The buffer must have a single word-long value for each map entry you set. For five color map entries, the color buffer must be five words long.

Only the eight most significant bits are used for color definition.

P3—Color buffer length, in bytes (required)

This longword contains the length of the color buffer, in bytes. The multiple to use depends on the system:

- Color—6
- Intensity—2

P4—Starting color map entry (must be 1)

This longword contains the color map index to use for the first RGB or intensity value specified in the buffer.

P6—Must be 0

DESCRIPTION

This function enables 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.

QDSS-Specific QIO Interface

Suspend Occluded Viewport Activity

Suspend Occluded Viewport Activity

Suspends all operations to a specified (occluded) viewport when certain operations are requested.

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

UNIQUE PARAMETERS **P1—IO\$C_QD_OCCLUDED_SUSPEND (required)**
This function code identifies the action the QIO performs.

P2—Viewport_id (required)
This parameter is the ID of the targeted viewport. The Get Viewport ID QIO returns the viewport ID during the creation of the viewport.

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 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 then issue a Resume Viewport Activity QIO to continue request queue processing. You can 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.

QDSS-Specific QIO Interface

Write Bitmap

Write Bitmap

Writes data from a user-specified buffer in VAX memory to a bitmap in video memory and performs bitmap-to-bitmap transfers.

FORMAT **SYSSQIO** *[efn] ,chan ,IO\$_QDWRITE ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2 ,[p3] ,p4 ,p5 [,p6]*

UNIQUE PARAMETERS

P1—Buffer address (required)

This parameter is the buffer address 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.

P2—Buffer length (required)

This longword contains the length of the buffer specified in P1, in bytes.

P3—Must be 0

P4—Transfer parameter block

This parameter is the transfer parameter block (TPB) that describes the bitmap to be written into video memory. The TPB contains:

- Coordinates for the lower left corner of the bitmap
- Height and width of the bitmap
- Predefined constant indicating the type of transfer
- For bitmap-to-bitmap transfer, the coordinates for the lower left corner of the target bitmap

The following diagram shows the data structure that specifies transfer parameters.

The following list describes the contents of each field in the Transfer Parameter block.

QDSS-Specific QIO Interface

Write Bitmap

Field	Use
TPB\$B_TYPE	Type of transfer being performed, either processor-to-bitmap (PTB) or bitmap-to-bitmap (BTB). Use the constants defined later to load this field.
TPB\$B_SIZE	Reserved to Digital
TPB\$W_X_SOURCE	X coordinate of lower left corner of source bitmap
TPB\$W_Y_SOURCE	Y coordinate of lower left 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 corner of target bitmap (Only specified for bitmap-to-bitmap transfer)
TPB\$W_Y_TARGET	Y coordinate of lower left 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

The following table defines the constants in conjunction with the TPB.

Constant	Value
TPB\$C_BITMAP_XFR	Bitmap-to-bitmap transfer
TPB\$C_SOURCE_ONLY	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

This longword contains the length of the transfer parameter block specified in P4, in bytes.

P6—Must be 0

DESCRIPTION

This function writes data from a specified buffer in VAX memory to QDSS video memory. This QIO 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.

An application can also use this function to perform bitmap-to-bitmap transfers in a bitmap-to-bitmap transfer, specifying a source and target location in the TPB and omitting the processor buffer.

5

Using Drawing Operation Primitives

This chapter includes the following topics:

- Overview of drawing primitives (DOPs)
- Drawing operations with DOPs
- Window management operations with DOPs
- UISDC interface features to use when implementing DOPs

5.1 Overview of DOPs

Drawing operation primitives (DOPs) are data structures created by your application that contain information the QDSS hardware uses to perform drawing operations on 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 and window management operations. Use DOPs to perform the following drawing operations:

- 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

Use DOPs to perform the following window management operations:

- Stop removing entries from a window request queue
- Start removing entries from a *stopped* request queue
- Suspend drawing operations to a window
- Resume drawing operations in a window

To perform a drawing operation, your application must complete the following steps:

- 1 Allocate storage for the DOP
- 2 Define the structure of the DOP

Using Drawing Operation Primitives

- 3 Initialize any relevant fields of the DOP structure
- 4 Execute the DOP

How you structure and initialize DOPs depends on which operation you perform.

How you allocate and execute DOPs also depends 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, as described in Section 5.4.

Sections 5.2 through 5.5 provide general information about DOPs that you must understand 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 DOP Structure

This section provides a general description of DOP structure. Section 5.6 gives a complete description of how to structure DOPs.

Each DOP structure consists of three substructures (blocks):

- **Common block**—This fixed-size block begins *all* DOP structures. It contains information that all DOPs require—for example, the *item_type* field that identifies which operation the DOP performs and the *opcount* field that indicates how many times the operation should be repeated.
- **Unique block**—This fixed-size block follows the Common block in all DOPs. It contains information more specific to a single operation, or group of operations, and its fields and their contents vary accordingly. Some operations do not use the fields in this block and others use only some of the fields (see the operation-by-operation structure description in Section 5.6 for details) but regardless of use, the DOP structure must be padded with the entire Unique block.
- **Variable block**—This variable-length block contains operation-specific variables (coordinates, line lengths, and so on). The size of this block depends on the number of times an operation is repeated. Thus, 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, the Variable block must hold the coordinates needed to draw three lines.

Section 5.6 contains an illustration of the Common block and a full explanation of each field in the block. The section also contains illustrations and explanations of the specific Unique and Variable blocks used to define DOPs for each type of operation.

5.3 Implementing DOPs in the UIS Environment

If your application runs in the UIS environment, you can use UISDC routines to allocate and execute DOPs.

5.3.1 Allocating Storage for DOPs in the UIS Environment

The `UISDC$ALLOCATE_DOP` routine allocates memory for the DOP and initializes some fields of the Common block to default values. `UISDC$ALLOCATE_DOP` has the following format:

```
dop_address = UISDC$ALLOCATE_DOP (wd_id, size, atb)
```

Where:

dop_address	Returned address of the DOP, used in subsequent routines to execute the DOP.
wd_id	Window identifier you specify that 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 <i>returned</i> size in subsequent operations.
atb	Address of an attribute block 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. Section 5.5.4.1 contains 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

```

Using Drawing Operation Primitives

5.3.1.1 Allocation Mechanism

To allocate DOPs, the system uses a mechanism that provides for efficient use of storage as follows:

- If possible, it allocates a small amount of storage.
- After DOP execution, it reuses the storage formerly occupied by DOPs.
- When too much storage is allocated, it waits for free memory.

By default, DOPs are allocated in two sizes:

- 128 bytes (small)
- 786 bytes (large)

You can also set DOP size and available number (see Section 5.3.1.2). When you specify the size of the variable portion of the DOP to allocate, the system determines whether it can allocate a small DOP, which it does 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, the system performs the following operations:

- Removes the DOP from the *request queue* (see Section 5.3.1.2)
- Puts the DOP on a *return queue*, which 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 use `UISDC$ALLOCATE_DOP` to allocate a DOP, the system completes the following steps:

- 1 Checks whether a small or large DOP is required
- 2 Attempts to reuse any previously executed DOPs of the same size
- 3 If there are no such DOPs, allocates a DOP from system memory or waits until a DOP is free for reuse

5.3.1.2 Modifying DOP Size and Number

The available default sizes and numbers of DOPs follow:

- 300 small DOPs, each 128 bytes long
- 150 large DOPs, each 768 bytes long

Use these values or alter them to suit your needs. If you want to change these values, do so before you attempt any other processing. That is, you cannot start with DOPs of one size then change the size and continue.

You can modify the default sizes of the small and large DOPs within the following restrictions:

- Small DOPs—128-512, inclusive
- Large DOPs—768-16384, inclusive

You can also modify the number of available DOPs, as long as you maintain a minimum of 110 small or 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 result in the need for more than 300 pages, take the following steps:

- 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.

Table 5-1 Redefinition of Logical Values

Logical	Default
<code>UIS\$SMALL_DOP_SIZE</code>	128
<code>UIS\$LARGE_DOP_SIZE</code>	768
<code>UIS\$NUMBER_OF_SMALL_DOPS</code>	300
<code>UIS\$NUMBER_OF_LARGE_DOPS</code>	150

5.3.2 Executing DOPs in the UIS Environment

Once you have defined, allocated, and initialized a DOP, you can execute it, with one of three options:

- 1 Use the `UISDC$EXECUTE_DOP_SYNC` routine to execute the DOP synchronously.
- 2 Use the `UISDC$EXECUTE_DOP_ASYNC` routine to execute the DOP asynchronously with completion notification.
- 3 Use the `UISDC$QUEUE_DOP` routine to execute the DOP asynchronously *without* completion notification.

The two types of asynchronous execution differ in that the `UISDC$EXECUTE_DOP_ASYNC` routine takes an I/O status block (IOSB) your application can use to check for completion notification. `UISDC$QUEUE_DOP` queues the DOP for execution but provides no way to tell when the operation is completed (it is more efficient). Complete descriptions of these routines and their arguments appear 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 FORTRAN program segment in Example 5-2 defines and initializes a DOP using a subroutine (the recommended method for FORTRAN) and queues a DOP for asynchronous execution without completion notification.

Using Drawing Operation Primitives

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                 ! DOP address, by value
                     %VAL(DOP))
```

5.3.2.3 Execution Mechanism

To execute DOPS, you use a mechanism called a *request queue*. Each viewport is associated with a request queue, which is a linked list of any DOPs submitted to a viewport for execution. Viewport IDs (including the systemwide viewport ID) are 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 where you want to execute the DOP (returned in the `UIS$CREATE_WINDOW` call) and the *address* of the DOP (returned by `UISDC$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 use the UIS windowing system (for example, if it provides its own windowing system), it must allocate and execute DOPs itself. Section 5.4.1 describes how an application allocates storage for DOPs. Section 5.4.2 describes how an application inserts DOPs on the request queue for execution.

5.4.1 Allocating Storage for DOPS in a Non-UIS Environment

Before you allocate new storage for a DOP, check the associated return queue to determine if any reusable storage is available there. Use the viewport ID to access the return queue (the address of the DOP Queue structure where the return queue resides). See Appendix B for a full description of the data structure.

Two return queues are associated with each viewport:

- One for large DOPs
- One for small DOPs

DOP size depends on the amount of information required to describe the requested drawing operations fully. Use the `MACRO REMQUE` instruction to remove a DOP from the return queue.

If no reusable storage exists, use the LIB\$GET_VM routine to allocate storage.

Example 5-3 creates a viewport, then allocates a DOP by first checking the return queue.

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_LINESC_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)

```

Example 5-3 Cont'd. on next page

Using Drawing Operation Primitives

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
```

Example 5-3 Cont'd. 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, you 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 (the address of the DOP Queue structure where the return queue resides). Appendix B describes the data structure.

Example 5-4 illustrates viewport creation, DOP allocation, loading the DOP with the necessary information, and DOP insertion on the request queue.

5.5 Structuring and Initializing DOPs

Each DOP structure consists of three blocks—Common block, Unique block, and Variable block. Common block fields are the same across operations; Unique and Variable block fields vary, depending on the operation.

Using Drawing Operation Primitives

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_LINESC_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

;++;
; 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
```

5.5.1 Common Block

The Common Block is a fixed-size block that begins *all* DOP structures. It contains a number of fields (described completely in the Common Block DOP Structure section), but two fields particularly affect the subsequent structure of the DOP:

- The *item_type* field
- The operation count *opcount* field

Item Type Field

The *item_type* field determines which operation the DOP performs and therefore affects which Unique and Variable blocks to specify. You must explicitly specify the item type with a predefined constant (defined in SYS\$LIBRARY:VWSSYSDEF). Table 5-2 lists and describes the symbolic constants used to specify all possible drawing operations.

Table 5-2 Symbolic Constants

Constant	Operation
DOP\$C_DRAW_LINES	Draws lines or polygon.
DOP\$C_DRAW_COMPLEX_LINE	Draws a complex patterned line with a sloped length and width. Also draws a filled polygon.
DOP\$C_DRAW_POINTS	Draws points.
DOP\$C_FILL_POLYGON	Draws a filled polygon with a specified pattern.
DOP\$C_FILL_POINT	Fills a point with a specified pattern.
DOP\$C_FILL_LINES	Fills a line with a specified pattern.
DOP\$C_DRAW_FIXED_TEXT	Draws text with a specified fixed-space font.
DOP\$C_DRAW_VAR_TEXT	Draws text with a specified variable-spaced font.
DOP\$C_MOVE_ROTATE_AREA	Moves an area with specified rotation and scaling.
DOP\$C_MOVE_AREA	Moves an area within a viewport.
DOP\$C_SCROLL_AREA	Moves area in a viewport. Fills the vacated area with background color.
DOP\$C_STOP	Stops removing entries from the request queue (used when manipulating the screen to handle occlusion).
DOP\$C_START	Starts removing entries from the specified request queue. This request can only be made from the system viewport.
DOP\$C_DELETE_BITMAP	Deletes an offscreen bitmap.
DOP\$C_SUSPEND	Stops removing DOPs from the request queue until the queue is resumed.
DOP\$C_RESUME	Resumes processing DOPs on the specified queue. This request can be made only from the system viewport.

Opcount Field

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 draws four lines. This field 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).

Using Drawing Operation Primitives

5.5.2 Unique Block

This fixed-size block contains information specific to a single operation or group of operations; its fields and their contents vary accordingly. Whether or not it uses the fields in the block, every DOP structure must include the Unique block. That is, if a particular operation uses only 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 appropriate for each operation.

5.5.3 Variable Block

This variable-sized block contains operation-specific variables such as coordinates and line lengths. Block size depends on the number of DOP operations. In other words, if you specify a draw-line operation with an *opcount* of one, the Variable block contains coordinates needed to draw one line. If you specify an *opcount* of three, the Variable block holds coordinates needed to draw three lines (and be three times as long).

Section 5.6 illustrates and explains the fields in the Variable block needed for a single occurrence of each operation. It also lists the predefined constant for the length of a one-operation Variable block. Use this constant to calculate the input size argument of the `UISDC$ALLOCATE_DOP` routine. Multiply the constant by the number of times the DOP is to repeat the operation (the *opcount*).

5.5.4 Programming Considerations

When you program an application that performs drawing operations, you can implement the DOP structure in various ways. Depending on the programming language and the exact nature of the operation, you can use one of these options:

- Use the predefined structure provided in the `SYSLIBRARY:VWSSYSDEF` file
- Define the full DOP structure using your language structured-type statements

If you use the predefined DOP structure, you do not have to construct the DOP explicitly in your application. However, it is not always possible to use the predefined structure.

5.5.4.1 The Predefined DOP Structure

SYSLIBRARY:VWSSYSDEF.*lan* (where *lan* is the file extension for your programming language) contains a DOP definition file.

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. The system allocates the DOP storage; you must associate a structure with the storage 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 that 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 the structure.

In each module where you reference it, you must "include" or "insert" the VWSSYSDEF file to use any predefined constants and offsets. Become familiar with the way the VWSSYSDEF file defines the DOP structure for your programming language.

Initializing fields with the offsets is straightforward with regard to the fixed portion of the DOP (the Common and Unique blocks) and somewhat more complex with regard to the Variable block. The VWSSYSDEF file defines offsets for only one occurrence of an operation in the Variable block from the end of the fixed portion. In other words, to draw a single line, the Variable block for a Draw Lines operation requires four fields with predefined offsets. To draw two lines, however, the Variable block requires eight fields, *but only four offsets are predefined*. To write to the second set of fields, use the same offsets *but change the point from which they are offset*.

In languages such as PASCAL that support arrays of structures, 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 such as FORTRAN that does *not* support arrays of structures, 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 use a loop that increments the offset to initialize the DOP for a series of operations.

For a one-operation case, FORTRAN can use the predefined offsets.

The examples that accompany the operation descriptions are written in FORTRAN. Section 5.5.4.2 describes the conventions they follow.

Using Drawing Operation Primitives

5.5.4.2 Using the Examples

The DOP reference section contains FORTRAN examples that are subroutines that construct the DOP structure and initialize necessary fields. To eliminate redundancy, the calling program is removed from each example. For the sake of illustration, Example 5-5 contains a sample calling program. The sample calling program performs the following functions:

- Creates a display and window.
- Allocates storage for a DOP that draws a box (four lines—opcount equals four).
- 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.)
- Passes the address of the Variable block (length of the fixed portion of the DOP plus 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: FORTRAN requires you 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 is not done in a subroutine, the RECORD statement allocates storage redundantly for the DOP.

5.6 The DOP Reference

This section provides an easy reference for structuring DOPs. It contains the following illustrations and descriptions:

- Illustration and detailed description of the the Common block structure
- Illustration and detailed description of the Unique and Variable block structure *for each operation*
- Description of how to perform each operation
- Program example of how to perform each operation

First the Common block is described, then each operation is described and 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. Each field lists the predefined field name. These names can be used to reference the DOPs.

NOTE: The descriptions assume that an application is using the UIS/UISDC interface to allocate and execute DOPs and load bitmaps. You can perform these functions without using the UIS/UISDC interface.

DOP Structures

Common Block

Common Block

All DOP structures must begin with the Common block.

common block (dop_structure)

Field	Description
DOP\$L_FLINK	DOP queue forward link, the address of the preceding DOP in the request queue
DOP\$L_BLINK	DOP queue backward link, the address of the subsequent DOP in the request queue
DOP\$W_SIZE	DOP size in bytes
DOP\$B_TYPE	DOP structure type
DOP\$B_SUB_TYPE	Reserved for use by Digital
DOP\$L_DEC_RESERVED	Reserved for use by Digital
DOP\$L_DEC_RESERVED2	Reserved for use by Digital
DOP\$L_USER_RESERVED	Reserved for use by the user
DOP\$W_FLAGS	<p>These are defined within <i>DOP\$W_FLAGS</i>:</p> <p>DOP\$V_DELETE_BITMAP This field is 1 bit long, and starts at bit 0. When this field equals 1, the offscreen source bitmap (identified in the <i>bitmap_ID</i> field) is deleted at operation completion.</p> <p>DOP\$V_SYSTEM_DOP This field is 1 bit long and starts at bit 1. When this field equals 1, the DOP is returned to the <i>system</i> 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 viewport.</p> <p>DOP\$V_NO_RETURN This field is 1 bit long and starts at bit 2. When this field equals 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.</p>
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).

DOP Structures Common Block

Field	Description
DOP\$W_ITEM_TYPE	Type of drawing operation requested by DOP. Symbolic constants used to specify all possible drawing operations follow: 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	The plane mask. This field should be -1 to be compatible with previous versions of VAX Workstation Software.
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 where 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.
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.

DOP Structures

Common Block

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 fields you must explicitly load depend on whether your application is running in the UIS environment.

If your application is not running in the UIS environment, it must load all relevant fields of the Common Block. That is, during allocation, the system loads all fields needed for the DOP except for the first six fields.

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 (default initialization); others must be explicitly initialized by your application (explicit initialization). The following sections list the fields in these two categories.

Default Initialization

Using the attribute block the application specifies in the routine call, UISDC\$ALLOCATE_DOP initializes the following fields:

- Writing mode
- Foreground and background colors
- Viewport minimum and maximum coordinates
- Clipping rectangle delta coordinates

You might affect any of these fields by modifying the ATB before allocation or by overriding initialization directly after allocation.

The ALLOCATE routine initializes the following fields, which are used for the DOP execution mechanism:

- Forward link
- Backward link
- Size
- Type
- Sub_type

You can access these fields, but do not attempt to modify them.

Explicit Initialization

Your application must explicitly initialize the following fields:

- *item_type*—Identifies which operation the DOP performs. To initialize this field, specify one of the symbolic constants listed above.

DOP Structures

Common Block

- *opcount*—Tells how many times the operation should be repeated. 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

Use the following fields in the Common block 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 to the foreground 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.

Writing mode determines 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). The 16 writing modes are described in Appendix C.

The source index is a number 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 following interactions:

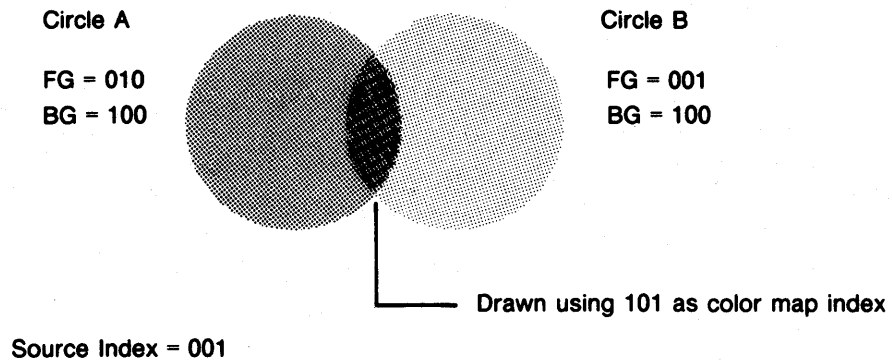
- State of the pixels in existing bitmap
- Specified source index value
- Specified writing mode
- Specified foreground and background colors.
- The *use_mask* modifier of the writing mode (if specified)
- Specified source bitmap (if *bitmap_id* field is specified)

Figure 5-1 demonstrates how the source index works. (This figure does not specify the source bitmap or the *use_mask* writing mode modifier.) It illustrates two intersecting circles: Circle A has a foreground index of 010 (binary—like all numbers in this example) and a background index of 100 (remember, the indexes represent colors in the color map—for simplicity, only three bits are used to represent pixel settings). Circle B has 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).

DOP Structures

Common Block

Figure 5-1 How the Source Index Works



MLO-1069-87

When you write objects 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 is specified in Figure 5-1, Circle A foreground index 010 is ORed with the source index, 001, resulting in the number 011.

The system uses the following procedure to determine which 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 that results 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.

Determining the Result of Screen Intersections

When you specify a bitmap (DOP\$L_BITMAP_ID is non-zero) and the use_mask_2 writing mode modifier, this process is more complicated. The following steps describe how to determine the result of any intersection on the screen. The sole objective is to overlay a character on top of an existing display. (In Figure 5-1, steps 2 and 5 are not meaningful because these options are not specified.)

STEP 1

```

bitmap_index
=
{
  IF      (source_bitmap exists AND source_bitmap = 0)
  THEN
    0
  ELSE
    source_index
}

```

DOP Structures Common Block

If you use the bitmap ID to specify a bitmap for the DOP, the driver creates a bitmap index by inserting the source index in any bitmap bit position where there is a 1. The *source_bitmap* is 1 bit; the resulting *bitmap_index* reflects the number of planes of color used.

STEP 2

$$\text{mask} = \left\{ \begin{array}{l} \text{IF} \\ \text{THEN} \\ \text{ELSE} \end{array} \left\{ \begin{array}{l} \text{use_mask} \\ \text{bitmap_index} \\ -1 \end{array} \right. \right\}$$

If you specify the *use_mask* modifier as part of the writing mode, the *bitmap_index* is used as the mask.

STEP 3

$$\text{fg_bg_selector} = \left\{ \begin{array}{l} \text{IF} \\ \text{THEN} \\ \text{ELSE} \end{array} \left\{ \begin{array}{l} \text{use_mask} \\ \text{(data) logical operation source_index} \\ \text{bitmap_index} \end{array} \right. \right\}$$

To obtain the foreground and background selector, perform a logical operation (determined by the specified writing mode) on the data on the screen and either the *source_index* value (if *use_mask* was specified) or the *bitmap_index*.

STEP 4

$$t1 = \left\{ \begin{array}{l} \text{(fg AND fg_bg_selector)} \\ \text{OR} \\ \text{(bg AND (NOT fg_bg_selector))} \end{array} \right\}$$

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 when no mask is specified.)

STEP 5

$$\text{final_data} = \left\{ \begin{array}{l} \text{(t1 AND mask)} \\ \text{OR} \\ \text{(data AND (NOT(mask)))} \end{array} \right\}$$

Use the mask and t1 to determine the data output to the screen.

Bitmap Fields

Two fields in the Common Block are used with bitmap operations: the *bitmap_ID* field and the *bitmap_glyphs* field. (Bitmap operations can be text or fill-pattern.)

An operation that uses a bitmap must first load the bitmap from processor memory into offscreen bitmap memory with the `UISDC$LOAD_BITMAP` routine. That routine returns a *bitmap_ID* that identifies the loaded bitmap in offscreen memory.

DOP Structures

Common Block

To use that bitmap in a bitmap-related DOP, you 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 uses the bitmap pattern to fill the polygon it creates.

The *bitmap_glyphs* field enables the system to retrieve the bitmap that is not available in offscreen memory. The *bitmap_glyph* is the address (in processor memory) where 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 one bit deep. Some operations (such as Move Area) might require a writing mode other than the default to operate properly.

By setting bits in the *flags* field, you can perform the following operations:

- Delete the bitmap specified in the *bitmap_ID* field, typically for a "one-shot" bitmap DOP. The DOP is performed, then the bitmap is deleted from offscreen memory.
- Not return a DOP for reuse, typically done for a "one-shot" DOP. This step prevents information contained in the DOP from being overwritten.
- Return the DOP storage to the *systemwide* return queue. This step is done when a DOP is allocated from the systemwide viewport and inserted on another viewport 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 corner of a bitmap in QDSS memory, specify 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

Since this is a queued operation, make sure that bitmap glyphs maintained by the application remain valid until Delete Bitmap has been executed.

Relevant Common Block Fields

Specify the bitmap ID of the bitmap you want 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 corner of a bitmap in QDSS memory, specify a for both the X and Y source coordinates.

Draw Complex Line

The Draw Complex Line operation enables 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)

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 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	X coordinate of the starting point of the line.
DOP_MOVE_R\$W_Y_TARGET	Y coordinate of the starting point of the line.
DOP_MOVE_R\$W_X_TARGET_VEC1	Delta of the X coordinate starting point and end point of vector 1.
DOP_MOVE_R\$W_Y_TARGET_VEC1	Delta of 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	Delta of the X coordinate starting point and end point of vector 2.
DOP_MOVE_R\$W_Y_TARGET_VEC2	Delta of 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 enables you to draw a line or variable width patterned line with a slope in both the length and width direction. Refer to 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 can 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 ones.

Initializing the Variable Block

To draw a complex line, specify a starting point and two vectors. Vector_1 indicates the length of the line and the slope of the length. Vector_2 indicates the width of the line and the slope of the width. 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 determine 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 corner of a bitmap in QDSS memory, specify 0 for both the X and Y source coordinates.

PERFORMANCE NOTE: Lines are drawn faster if Vector_1 specifies a slope of less than 45 degrees.

EXAMPLE

The following FORTRAN program draws a complex line to the screen as follows:

- 1 Creates a bitmap pattern
- 2 Uses UISDC\$LOAD_BITMAP to load the bitmap
- 3 Specifies an opcount of 1
- 4 Initializes the Variable block with the offset and extent of the specified bitmap pattern
- 5 Initializes the Variable block with a starting position of (50,50)
- 6 Initializes 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)
```

DOP Structures

Draw Complex Line

```

! 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

! 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

This operation describes the additional DOP structure needed to draw fixed-width text to the screen.

unique block (text_args)

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)

Field	Use
DOP_FTEXT\$W_OFFSET_X	X coordinate of the offset into the font where a specific character is located.
DOP_FTEXT\$W_OFFSET_Y	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 enables 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, first load a bitmap that contains a fixed-width font from processor memory to the offscreen bitmap memory. Use the UISDC\$LOAD_BITMAP routine to load a bitmap. This routine returns a bitmap ID that must be loaded in the *bitmap_ID* field of the Common block. (Section 5.7 describes 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 you write text, you specify the starting point as the *upper* left corner of the first character (provided the character height is specified as a positive number).

DOP Structures

Draw Fixed Text

Initializing the Variable Block

The Variable block specifies an offset into the font to indicate which character to write.

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 corner of a bitmap in QDSS memory, specify 0 for both the X and Y source coordinates.

EXAMPLE

The following FORTRAN program writes two characters of text to the screen as follows steps:

- 1 Creates a two-letter bitmap—H I (using a function)
- 2 Uses UISDC\$LOAD_BITMAP to load the bitmap
- 3 Specifies an opcount of 2
- 4 Initializes the Unique block with a starting position of (100,100)
- 5 Initializes the Variable block with the offsets of the two characters

Note that the specified offset values seem to write the letters in reverse order 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)
```

DOP Structures Draw Fixed Text

```
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
```

DOP Structures

Draw Lines

Draw Lines

This operation describes the additional structure needed to draw lines with specified end points.

unique block (plot_args)

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)

Field	Use
DOP_LINE\$W_X1	X coordinate of the starting point of the line
DOP_LINE\$W_Y1	Y coordinate of the starting point of the line
DOP_LINE\$W_X2	X coordinate of the end point of the line
DOP_LINE\$W_Y2	Y coordinate of the end point of the line

Variable-Length Constant: dop_line\$c_length

DESCRIPTION The Draw Lines operation enables you to draw the following structures:

- 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 with the bitmap offset specified (if specified). Fill Lines begins drawing at the specified starting point revealing the repeated fill pattern 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 can 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 ones.

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 its 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 want 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 corner of a bitmap in QDSS memory, specify 0 for both the X and Y source coordinates.

EXAMPLE

This FORTRAN program performs 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
```

DOP Structures

Draw Lines

```
      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

This operation describes the additional DOP structure you need to draw points to the screen.

unique block (plot_args)

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)

Field	Use
DOP_POINT\$W_X	X coordinate of the point to draw
DOP_POINT\$W_Y	Y coordinate of the point to draw

Variable-Length Constant: dop_draw_points\$c_length

DESCRIPTION

The Draw Points operation enables 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

If you specify a bitmap ID in the Common block to indicate a pattern to use when drawing a point or a series of points, use the following fields:

- *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
- *width* and *height* fields—To specify the extents of the bitmap

The pattern is incremented one pixel/point and repeats when the width is reached. This operation is useful for drawing a thin patterned curve.

By default, the system uses a fill pattern of all ones.

DOP Structures

Draw Points

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 want 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 corner of a bitmap in QDSS memory, specify 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

This operation describes the additional DOP structure needed to draw text of variable width to the screen.

unique block (text_args)

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)

Field	Use
DOP_FTEXT\$W_OFFSET_X	X coordinate of the offset into the font where a specific character is located
DOP_FTEXT\$W_OFFSET_Y	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 enables 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, first load a bitmap with a variable-width font from processor memory to the offscreen bitmap memory. To load a bitmap, use 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 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 you write text, the position you specify as the starting point is the *upper* left corner of the first character (provided the character height is specified as a positive number).

Initializing the Variable Block

The Variable block indicates which character to write as follows:

- Specifies an offset into the font
- 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 corner of a bitmap in QDSS memory, specify 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 Uses a function to creates a 3-letter bitmap—P L I
- 2 Uses UISDC\$LOAD_BITMAP to load the 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 because of the way VAX memory loads the bitmap, the specified offset values appear reversed: the I is loaded at (0,0).

DOP Structures

Draw Variable Text

```

! 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 Structures Draw Variable Text

```
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
```

DOP Structures

Fill Lines

Fill Lines

This operation describes the additional structure needed to draw lines, using a specified bitmap pattern.

unique block (plot_args)

Field	Use
DOP\$W_PLOT_FILL_WIDTH	Width of fill pattern (from offscreen bitmap), in bits; bitmap is identified in <i>bitmap_ID</i> field
DOP\$W_PLOT_FILL_HEIGHT	Height of fill pattern (from offscreen bitmap), in bits; bitmap is identified in <i>bitmap_ID</i> field
DOP\$W_PLOT_FILL_PATTERN_X	X coordinate in the bitmap from which to base the fill pattern; bitmap is identified in <i>bitmap_ID</i> field
DOP\$W_PLOT_FILL_PATTERN_Y	Y coordinate in the bitmap from which to base the fill pattern; bitmap is identified in <i>bitmap_ID</i> field

variable block (dop_line_array)

Field	Use
DOP_LINE\$W_X1	X coordinate of the starting point of the line
DOP_LINE\$W_Y1	Y coordinate of the starting point of the line
DOP_LINE\$W_X2	X coordinate of the end point of the line
DOP_LINE\$W_Y2	Y coordinate of the end point of the line

Variable-Length Constant: *dop_line\$c_length*

DESCRIPTION

The Fill Lines operation enables 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 might be lifted in a future release.

Relevant Common Block Fields

To draw a patterned line to the screen, use the `UISDC$LOAD_BITMAP` routine to load a bitmap with the pattern from processor memory to the offscreen bitmap memory. 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 the portion of the loaded bitmap pattern used to fill the line.

Initializing the Variable Block

To draw a line, specify the line 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 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 corner of a bitmap in QDSS memory, specify 0 for both the X and Y source coordinates.

EXAMPLE

This FORTRAN example program draws a filled line as follows:

- 1 Uses UISDC\$LOAD_BITMAP to load a bitmap pattern
- 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

```

DOP Structures

Fill Lines

```
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_LINES$W_X1 = 50
DOP_VAR.DOP_LINES$W_Y1 = 50
DOP_VAR.DOP_LINES$W_X2 = 150
DOP_VAR.DOP_LINES$W_Y2 = 50

RETURN
END
```

Fill Point

This operation describes the additional DOP structure needed to map a point to a defined bitmap.

unique block (plot_args)

Field	Use
DOP\$W_PLOT_FILL_WIDTH	Width of fill pattern (from offscreen bitmap), in bits; bitmap is identified in <i>bitmap_ID</i> field
DOP\$W_PLOT_FILL_HEIGHT	Height of fill pattern (from offscreen bitmap), in bits; bitmap is identified in <i>bitmap_ID</i> field
DOP\$W_PLOT_FILL_PATTERN_X	X coordinate in the bitmap from which to base the fill pattern; bitmap is identified in <i>bitmap_ID</i> field
DOP\$W_PLOT_FILL_PATTERN_Y	Y coordinate in the bitmap from which to base the fill pattern; bitmap is identified in <i>bitmap_ID</i> field

variable block (dop_point_array)

Field	Use
DOP_POINT\$W_X	X coordinate of the point to fill
DOP_POINT\$W_Y	Y coordinate of the point to fill

DESCRIPTION

The Fill Point operation enables you to map individual points you draw to the screen to a bitmap pattern you specify. If the corresponding bit in the bitmap is set, 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, first load a bitmap that contains the pattern from processor memory to the offscreen bitmap memory. Use the `UISDC$LOAD_BITMAP` routine to load a bitmap. 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.

DOP Structures

Fill 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 corner of a bitmap in QDSS memory, specify 0 for both X and Y source coordinates.

EXAMPLE

This sample FORTRAN program segment draws 90 points in a patterned sine curve as follows:

- 1 Creates a bitmap pattern (in a function)
- 2 Uses `UISDC$LOAD_BITMAP` to load the bitmap
- 3 Specifies an `opc` count 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

DOP Structures

Fill Point

```

! 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

```

DOP Structures

Fill Point

```
! 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

This operation describes the additional DOP structure to create a polygon and fill it with a specified pattern.

unique block (plot_args)

Field	Use
DOP\$W_PLOT_FILL_WIDTH	Width of fill pattern (from offscreen bitmap), in bits; bitmap is identified in <i>bitmap_ID</i> 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 <i>bitmap_ID</i> field. If no bitmap is specified, this field is ignored.
DOP\$W_PLOT_FILL_PATTERN_X	X coordinate in the bitmap from which to base the fill pattern; bitmap is identified in <i>bitmap_ID</i> field. If no bitmap is specified, this field is ignored.
DOP\$W_PLOT_FILL_PATTERN_Y	Y coordinate in the bitmap from which to base the fill pattern; bitmap is identified in <i>bitmap_ID</i> field. If no bitmap is specified, this field is ignored.

variable block (dop_poly_array)

Field	Use
DOP_POLY\$W_LEFT_X1	X coordinate of the starting point of a line that defines the left edge of a polygon
DOP_POLY\$W_LEFT_Y1	Y coordinate of the starting point of a line that defines the left edge of a polygon
DOP_POLY\$W_LEFT_X2	X coordinate of the end point of a line that defines the left edge of a polygon
DOP_POLY\$W_LEFT_Y2	Y coordinate of the end point of a line that defines the left edge of a polygon
DOP_POLY\$W_RIGHT_X1	X coordinate of the starting point of a line that defines the right edge of a polygon
DOP_POLY\$W_RIGHT_Y1	Y coordinate of the starting point of a line that defines the right edge of a polygon
DOP_POLY\$W_RIGHT_X2	X coordinate of the end point of a line that defines the right edge of a polygon
DOP_POLY\$W_RIGHT_Y2	Y coordinate of the end point of a line that defines the right edge of a polygon

Variable-Length Constant: dop_poly\$c_length

DOP Structures

Fill Polygon

DESCRIPTION

The Fill Polygon operation enables you to write a polygon (more precisely, a trapezoid) 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, Y coordinates of the upper corners must be the same, and Y coordinates of the lower corners must be the same.

NOTE: Some of these restrictions might be lifted in a future release.

Relevant Common Block Fields

To draw a pattern-filled polygon to the screen, first use the `UISDC$LOAD_BITMAP` routine to load a bitmap with the desired pattern from processor memory to the offscreen bitmap memory. 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 you should use 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 (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, Y coordinates of the upper corners must be the same, and 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 corner of a bitmap in QDSS memory, specify 0 for both the X and Y source coordinates.

EXAMPLE

This FORTRAN sample program segment draws a filled polygon as follows:

- 1 Creates a bitmap pattern (in a function)
- 2 Uses `UISDC$LOAD_BITMAP` to load the 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
```


DOP Structures Fill Polygon

```
! 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 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
```

DOP Structures Fill Polygon

RETURN
END

Move Area

This operation 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)

Field	Use
DOP_MOVE\$W_FILLER	Reserved for use by DIGITAL
DOP_MOVE\$W_X_SOURCE	X coordinate of the lower left corner of the source area (area to be moved)
DOP_MOVE\$W_Y_SOURCE	Y coordinate of the lower left 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	X coordinate of the lower left corner of the target area (the area to which the source area is moved)
DOP_MOVE\$W_Y_TARGET	Y coordinate of the lower left corner of the target area

Variable-Length Constant: dop_move\$c_length

DESCRIPTION

The Move Area operation enables you to move (copy) a rectangular area from one point to another, either on the screen or in offscreen memory.

Relevant Common Block Fields

Overlay is the default writing mode loaded into the Common block (from the ATB) during allocation. When you are performing a Move Area operation, use *COPY mode*. To load the UIS\$MODE_COPY value into the *writing_mode* field of the Common block, use the UIS\$SET_WRITING_MODE routine to modify the ATB before you allocate, or use the predefined offsets to load the field with the value directly.

WRIT\$C_S (source only) is the non-UIS environment writing mode that corresponds to COPY.

Initializing the Variable Block

To move an area, specify the coordinates that define the lower left corner of the rectangle you want to move (source area), the height and width of the area, and the coordinates that define the lower left corner of the area where you are moving it (target area).

DOP Structures

Move 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) because when text is written to a specified point on the screen (that is, 100,100) the point is considered the *upper* left corner of the text—but for Move Area you must specify the *lower* left corner of the rectangle.

The full calling program is shown to clarify the example.

```
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
```

DOP Structures Move Area

```

! 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 *
! *****

```

DOP Structures

Move Area

```
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$W_Y_SOURCE = 93
  DOP_VAR.DOP_MOVE$W_WIDTH = 16
  DOP_VAR.DOP_MOVE$W_HEIGHT = 8
  DOP_VAR.DOP_MOVE$W_X_TARGET = 50
  DOP_VAR.DOP_MOVE$W_Y_TARGET = 50
  !text is written in
  !negative direction
  RETURN
END
```

Move/Rotate Area

This operation describes the additional DOP structure needed 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)

Field	Use
DOP_MOVE_R\$W_FILLER	Reserved for use by Digital
DOP_MOVE_R\$W_X_SOURCE	X coordinate of the lower left 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	Y coordinate of the lower left 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	X coordinate of the lower left corner of the target area (the area to which the source area is moved)
DOP_MOVE_R\$W_Y_TARGET	Y coordinate of the lower left corner of the target area
DOP_MOVE_R\$W_X_TARGET_VEC1	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	Y coordinate of the end point of vector 1
DOP_MOVE_R\$W_L_TARGET_VEC1	Length of vector 1; determines the degree of scaling
DOP_MOVE_R\$W_X_TARGET_VEC2	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	Y coordinate of the end point of vector 1
DOP_MOVE_R\$W_L_TARGET_VEC2	Length of vector 2; determines the degree of scaling

DESCRIPTION

DOP Structures

Move/Rotate Area

The Move Rotate Area operation enables you to move (copy) a rectangular area from one point to another (either on the screen or in offscreen memory) and enables you to rotate and scale the moved copy.

This operation also enables 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 *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. Typically, you use this feature to move, rotate, and/or scale text.

The default writing mode loaded into the Common block (from the ATB) during allocation is *overlay mode*. When you move areas, use *COPY mode*. To load the `UIS$_MODE_COPY` value into the *writing mode* field of the Common block, either use the `UIS$SET_WRITING_MODE` routine to modify the ATB before allocating or use the predefined offsets to load the field with the value directly. `WRIT$_C_S` (source only) is the non-UIS environment writing mode that corresponds to *COPY*.

Initializing the Variable Block

To move an area, you must specify the coordinates that define the lower left corner of the rectangle you want to move (source area), the height and width of the area, and the coordinates that define the lower left corner of the area where you are moving it (target area).

To rotate and scale a moved area, specify two vectors, each with the following values:

- X and Y value—Specify angle of rotation
- Length value—Specifies degree of scaling

One vector relates 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 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)
```

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 is 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 corner of a bitmap in QDSS memory, specify 0 for both X and Y source coordinates.

EXAMPLE

The following FORTRAN program draws an 11- 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 SYSSHIBER()

END

! *****
! * DRAW_LINES SUBROUTINE *
! *****

SUBROUTINE D_LINES (DOP, DOP_VAR)

```

DOP Structures

Move/Rotate Area

```

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 Structures Move/Rotate Area

```
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
```

DOP Structures

Resume Viewport Activity

Resume Viewport Activity

This operation describes the additional DOP structure needed for a system viewport to resume activity on a suspended viewport.

unique block (stop_args)

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 with the Suspend Viewport Activity DOP (or Suspend Viewport Activity QIO function).

Because 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 corner of a bitmap in QDSS memory, specify 0 for both 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
```

DOP Structures

Resume Viewport Activity

RETURN
END

DOP Structures

Scroll Area

Scroll Area

This operation 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)

Field	Use
DOP_MOVE\$W_FILLER	Reserved for use by Digital
DOP_MOVE\$W_X_SOURCE	X coordinate of the lower left corner of the source area (area to be moved)
DOP_MOVE\$W_Y_SOURCE	Y coordinate of the lower left 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	X coordinate of the lower left corner of the target area (the area to which the source area is moved)
DOP_MOVE\$W_Y_TARGET	Y coordinate of the lower left 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, specify the coordinates that define the lower left corner of the rectangle you want to move (source area), the height and width of the area, and the coordinates that define the lower left corner of the area where you are moving it (target area).

EXAMPLE

The following FORTRAN program scrolls an 11- 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_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 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/

```

DOP Structures

Scroll Area

```
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

DOP Structures

Start Request Queue

Start Request Queue

This operation describes the additional DOP structure needed for the system viewport to restart a *stopped* request queue on another viewport.

unique block (stop_args)

Field	Use
DOP\$L_DRIVER_VP_ID	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. 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 corner of a bitmap in QDSS memory, specify 0 for both 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
```

DOP Structures

Start Request Queue

RETURN
END

Stop Request Queue

This operation describes the additional DOP structure needed to stop removing entries from the specified viewport request queue.

unique block (stop_args)

Field	Use
DOP\$L_DRIVER_VP_ID	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 request queue ensures that no other process will modify the bitmap of the stopped viewport. To guarantee that all previously queued DOPs are processed, insert this DOP on the queue with the Insert DOP QIO or the Execute DOP UISDC routine. Stop differs from Suspend in 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 (with 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 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. 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 corner of a bitmap in QDSS memory, specify 0 for both 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
```

DOP Structures

Suspend Viewport Activity

Suspend Viewport Activity

This operation describes the additional DOP structure to suspend activity on a viewport.

unique block
(stop_args)



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 issuing a \$QIO request to resume viewport activity (see Chapter 4) or invoking the Resume DOP (from a viewport other than the stopped one). Think of Suspend and Resume as the drawing parallels to the CTRL/S and CTRL/Q key functions.

Stop differs from Suspend in 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. 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 corner of a bitmap in QDSS memory, specify 0 for both 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 enables applications that draw within UIS viewports to use DOPs. The five UISDC routines that compose the interface follow.

- `UISDC$ALLOCATE_DOP`—Allocate storage for a DOP
- `UISDC$LOAD_BITMAP`—Load bitmaps from processor memory into offscreen bitmap memory (for subsequent use by text or fill pattern DOPs)
- `UISDC$EXECUTE_DOP_ASYNCH`, `UISDC$EXECUTE_DOP_SYNC`, and `UISDC$QUEUE_DOP`—Submit DOPs to the request queue for execution

This section describes each routine.

Section 5.3.1 describes how to use `UISDC$ALLOCATE_DOP` to allocate DOPs.

Section 5.3.2 describes how to use `UISDC$EXECUTE_DOP_ASYNCH`, `UISDC$EXECUTE_DOP_SYNC`, and `UISDC$QUEUE_DOP` to execute DOPs (and how they differ from one another).

Section 5.7.1 describes how to use `UISDC$LOAD_BITMAP`.

5.7.1 Loading Bitmaps into Offscreen Memory

Use the `UISDC$LOAD_BITMAP` routine to load a bitmap. `UISDC$LOAD_BITMAP` returns a bitmap identifier (a handle) when you specify a window ID (where 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).

Use `UISDC$LOAD_BITMAP` two ways.

- 1 The bitmap is copied from the user buffer into a driver-maintained buffer. When the bitmap is accessed, it is copied from the driver-maintained buffer into offscreen memory.
- 2 A handle (identifier) is created for the bitmap, but the routine 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.

Using Drawing Operation Primitives

To load a bitmap dynamically, 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).

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 Routines

UISDC\$LOAD_BITMAP

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 wraps. 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.

DESCRIPTION UISDC\$EXECUTE_DOP_ASYNC queues the specified DOP for execution in the specified window request queue. The execution is performed asynchronously; the DOP is queued for execution, but control is immediately returned to the application. Use the IOSB to determine when the DOP has actually completed.

UISDC\$QUEUE_DOP—Queue Drawing Operation Primitive

This routine queues the specified DOP 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 *VMS Workstation Software 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**

Drawing Operation Primitive. The *dop_address* argument is the address of an array of bytes that compose 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 request queue. The execution is performed asynchronously; the DOP is queued for execution, but control is immediately returned to the application. This differs from *UISDC\$EXECUTE_DOP_ASYNC* in that with *UISDC\$QUEUE_DOP*, you *cannot* determine when the DOP has actually completed.

A

QVSS/QDSS Data Structures

This appendix illustrates and describes the data structures common to the QVSS and QDSS systems.

button simulation block

buttons to be pressed mask	0
buttons to be released mask	4
0	8
0	12

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.										
	Pointer button definitions used in the masks are defined by the \$QVBDEF macro. They consist of the following symbols:										
	<table border="1"> <thead> <tr> <th>Symbol</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_BUTTON_1</td> <td>Select button</td> </tr> <tr> <td>QV\$M_BUTTON_2</td> <td>Button 2</td> </tr> <tr> <td>QV\$M_BUTTON_3</td> <td>Button 3</td> </tr> <tr> <td>QV\$M_BUTTON_4</td> <td>Button 4</td> </tr> </tbody> </table>	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
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	0
Y offset	4

QVSS/QDSS Data Structures

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	X offset from the upper left corner of the pointer pattern to the active point.
Y offset	Y offset from the upper left corner of the pointer pattern to the active point.

data rectangle values block

MINX (left side value)	0
MINY (bottom side value)	4
MAXX (right side value)	8
MAXY (top side value)	12

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	0
AST parameter	4
access mode	8
0	12

Field	Use
AST service routine address	The AST service routine address is 0 if no AST action 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 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

enabled characteristics mask	0
disabled characteristics mask	4
keyclick volume	8
0	12

Field	Use
Enabled characteristics mask	The first longword is a mask of characteristics to be enabled.

QVSS/QDSS Data Structures

Field	Use																														
Disabled characteristics mask	<p>The second longword is a mask of characteristics to be disabled.</p> <p>The keyboard characteristics, defined by the \$QVBDEF macro, consist of the following bits:</p> <table border="1"> <thead> <tr> <th>Characteristic</th> <th>Default</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_KEY_AUTORPT</td> <td>On</td> <td>Key held down automatically repeats.</td> </tr> <tr> <td>QV\$M_KEY_KEYCLICK</td> <td>On</td> <td>Keyclick sounds on each keystroke.</td> </tr> <tr> <td>QV\$M_KEY_UDF6</td> <td>Off</td> <td>Function keys F6 through F10 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF11</td> <td>Off</td> <td>Function keys F11 through F14 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDF17</td> <td>Off</td> <td>Function keys F17 through F20 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDHELPDO</td> <td>Off</td> <td>Function keys HELP and DO generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDE1</td> <td>Off</td> <td>Function keys E1 through E6 generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDARROW</td> <td>Off</td> <td>Arrow keys generate up/down transitions.</td> </tr> <tr> <td>QV\$M_KEY_UDNUMKEY</td> <td>Off</td> <td>Numeric keypad keys generate up/down transitions.</td> </tr> </tbody> </table>	Characteristic	Default	Meaning	QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.	QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.	QV\$M_KEY_UDF6	Off	Function keys F6 through F10 generate up/down transitions.	QV\$M_KEY_UDF11	Off	Function keys F11 through F14 generate up/down transitions.	QV\$M_KEY_UDF17	Off	Function keys F17 through F20 generate up/down transitions.	QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.	QV\$M_KEY_UDE1	Off	Function keys E1 through E6 generate up/down transitions.	QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.	QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.
Characteristic	Default	Meaning																													
QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.																													
QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.																													
QV\$M_KEY_UDF6	Off	Function keys F6 through F10 generate up/down transitions.																													
QV\$M_KEY_UDF11	Off	Function keys F11 through F14 generate up/down transitions.																													
QV\$M_KEY_UDF17	Off	Function keys F17 through F20 generate up/down transitions.																													
QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.																													
QV\$M_KEY_UDE1	Off	Function keys E1 through E6 generate up/down transitions.																													
QV\$M_KEY_UDARROW	Off	Arrow keys generate up/down transitions.																													
QV\$M_KEY_UDNUMKEY	Off	Numeric keypad keys generate up/down transitions.																													
Keyclick volume	<p>The keyclick volume is a value from 1 (loudest) to 8 (softest). If a value of 0 is specified, the current system default keyclick volume is used.</p>																														
0	<p>The fourth longword must be 0.</p>																														

**keystroke ast
specification
block**

AST service routine address	0
AST parameter	4
access mode	8
input token address	12

Field	Use
AST service routine address	The AST service routine address is 0 if no AST action 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 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 from 0 to 255 map into the Digital multinational character set. Values from 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	0
disabled characteristics mask	4
0	8
0	12

Field	Use						
Enabled characteristics mask	Longword of characteristics to be enabled.						
Disabled characteristics mask	Longword of characteristics to be disabled.						
	The pointer button characteristics, defined by the \$QVBDEF macro, consist of the following bits:						
	<table border="1" style="width: 100%;"> <thead> <tr> <th style="text-align: left;">Characteristic</th> <th style="text-align: left;">Default</th> <th style="text-align: left;">Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_BUT_UPTODOWN</td> <td>On</td> <td>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 the active button request for the current pointer cursor position.</td> </tr> </tbody> </table>	Characteristic	Default	Meaning	QV\$M_BUT_UPTODOWN	On	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 the active button request for the current pointer cursor position.
Characteristic	Default	Meaning					
QV\$M_BUT_UPTODOWN	On	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 the active button request for the current pointer cursor position.					
0	This longword must be zero.						
0	This longword must be zero.						

pointer characteristics block

enabled characteristics mask	0
disabled characteristics mask	4
0	8
0	12

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, defined by the \$QVBDEF macro, consist of the following bits:						
	<table border="1"> <thead> <tr> <th>Characteristic</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>QV\$M_PTR_LEFT_HAND</td> <td>Invert buttons on mouse or puck. (Buttons 1 and 3 are switched.)</td> </tr> <tr> <td>QV\$M_PTR_INVERT_STYLUS</td> <td>Invert buttons on stylus. (Buttons 1 and 3 are switched.)</td> </tr> </tbody> </table>	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.)
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**

AST service routine address	0
AST parameter	4
access mode	8
address of new pointer cursor position	12

Field	Use
AST service routine address	AST service routine address is 0 if no AST action routine is required. No buffering of data in the type-ahead buffer occurs for pointer motion ASTs.

QVSS/QDSS Data Structures

Field	Use
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 value 0 through 1023 with the lowest value denoting the left side of the screen; Y is value 0 through 863 with the lowest value denoting the bottom of the screen. For tablet pointers, the range of X is defined in the <code>rqvbw_tablet_width</code> field of the QVSS block; the range of Y is defined in the <code>qvbw_tablet_height</code> field of the QVSS block.

new cursor position

X position on physical screen	0
Y position on physical screen	4

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	0
Y position on physical screen	4

Field	Use
X position on the physical screen	X position on the physical screen

Field	Use
Y position on the physical screen	Y position on the physical screen

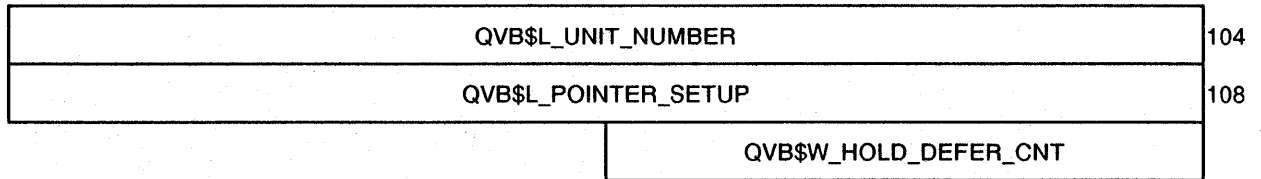
QVSS/QDSS Data Structures

qvss block (qvb) (qvb_common_structure)

QVB\$L_VIDEOSIZE			0
QVB\$L_VIDEOADDR			4
QVB\$L_MAPSIZE			8
QVB\$L_MAPADDR			12
QVB\$L_CONTEXT			16
QVB\$L_CSR			20
QVB\$W_MOUS_YPIX	QVB\$W_MOUS_XPIX		24
QVB\$W_HEIGHT	QVB\$W_WIDTH		28
QVB\$W_Y_RESOL	QVB\$W_X_RESOL		32
QVB\$L_MOUS_XABS			36
QVB\$L_MOUS_YABS			40
QVB\$L_MAIN_VIDEOSIZE			44
QVB\$L_MAIN_VIDEOADDR			48
QVB\$L_MAIN_MAPSIZE			52
QVB\$L_MAIN_MAPADDR			56
QVB\$W_MAIN_MAPMAX	QVB\$W_MAIN_MAPMIN		60
QVB\$L_CHARACTERISTICS			64
QVB\$W_BUTTONS	QVB\$W_SCRSAV_TIMEOUT		68
QVB\$W_TABLET_XPIX	QVB\$W_KEYCLICK_VOLUME		72
QVB\$W_TABLET_WIDTH	QVB\$W_TABLET_YPIX		76
QVB\$W_BUT_STATUS	QVB\$W_TABLET_HEIGHT		80
QVB\$W_FLAGS	DEVICE_TYPE	BITS_PER_PIXEL	84
QVB\$W_SPARE_W_1	SPARE_B_1	CURSOR_PLANES	88
QVB\$W_TABLET_YSIZE	QVB\$W_TABLET_XSIZE		92
QVB\$F_TABLET_XRATIO			96
QVB\$F_TABLET_YRATIO			100

(Continued on next page)

Example 5-5 (Cont.) Calling Program for Example Subroutines



The following list describes the contents of each field in the QVSS block.

NOTE: The names in the following fields of the preceding data structure have the prefix QVB\$B_. The prefixes are omitted in the diagram so the field names 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.
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).

QVSS/QDSS Data Structures

Field	Use						
QVB\$_MAIN_MAPADDR	Address of the windowing system scanline map (QVSS specific)						
QVB\$_MAIN_MAPMIN	Entry number of the lowest entry in the scanline map last updated (QVSS specific).						
QVB\$_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\$_CHARACTERISTICS	Current systemwide windowing characteristics.						
QVB\$_SCRSAV_TIMEOUT	Current screen saver timeout value, in seconds.						
QVB\$_BUTTONS	This field no longer supported.						
QVB\$_KEYCLICK_VOLUME	Default keyclick volume. Value must be in the range of 1 to 8 (1 is loudest). Default is 3.						
QVB\$_TABLET_XPIX	Current X position on a tablet, in pixels.						
QVB\$_TABLET_YPIX	Current Y position on a tablet, in pixels.						
QVB\$_TABLET_WIDTH	Maximum horizontal size of a tablet, in pixels.						
QVB\$_TABLET_HEIGHT	Maximum vertical size of a tablet, in pixels.						
QVB\$_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\$_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\$_DEVICE_TYPE	Value identifying driver: 0 for QVSS, 1 for QDSS.						
QVB\$_FLAGS	Internal flags. The following fields are defined in QVB\$_FLAGS :						
	<table border="1"> <thead> <tr> <th>Field</th> <th>Use</th> </tr> </thead> <tbody> <tr> <td>QVB\$_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\$_STYLUS</td> <td>This field is one bit long and starts at bit 19; indicates that tablet stylus is present.</td> </tr> </tbody> </table>	Field	Use	QVB\$_TABLET	This field is one bit long and starts at bit 18; indicates that tablet is present; 0 equals pointer is present.	QVB\$_STYLUS	This field is one bit long and starts at bit 19; indicates that tablet stylus is present.
Field	Use						
QVB\$_TABLET	This field is one bit long and starts at bit 18; indicates that tablet is present; 0 equals pointer is present.						
QVB\$_STYLUS	This field is one bit long and starts at bit 19; indicates that tablet stylus is present.						
QVB\$_CURSOR_PLANES	The number of planes in the hardware cursor. A color cursor has two planes.						
QVB\$_SPARE_B_1	Reserved to Digital.						
QVB\$_SPARE_W_1	Reserved to Digital.						
QVB\$_TABLET_XSIZE	Width of tablet, in centimeters (integer value).						

QVSS/QDSS Data Structures

Field	Use						
QVB\$W_TABLET_YSIZE	Height of tablet, in centimeters (integer value).						
QVB\$F_TABLET_XRATIO	Floating-point ratio of screen width to tablet width, in pixels.						
QVB\$F_TABLET_YRATIO	Floating-point ratio of screen height to tablet height, in pixels.						
QVB\$L_UNIT_NUMBER	Number of the video device unit associated with the QVB.						
QVB\$L_POINTER_SETUP	System characteristics. Contains the current status for system wide pointer characteristics. The following fields are defined within QVB\$L_POINTER_SETUP:						
	<table border="1"> <thead> <tr> <th>Field</th> <th>Use</th> </tr> </thead> <tbody> <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> </tbody> </table>	Field	Use	QV\$V_PTR_LEFT_HAND	Indicates pointer is left handed if bit is set.	QV\$V_PTR_INVERT_STYLUS	Indicates tip of stylus is select button if bit is set.
Field	Use						
QV\$V_PTR_LEFT_HAND	Indicates pointer is left handed if bit is set.						
QV\$V_PTR_INVERT_STYLUS	Indicates tip of stylus is select button if bit is set.						
QVB\$W_HOLD_DEFER_CNT	Reserved for use by Digital.						

The following constants are defined in conjunction with the QDB\$.

Constant	Value
KEY\$C_SELECT	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	Length of the QVB structure

QVSS/QDSS Data Structures

reserved function keystroke ast specification block

AST service routine address	0
AST parameter	4
access mode	8
input token address	12

Field	Use
AST service routine address	AST service routine address 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	This address of a longword 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)	0
MINY (bottom side value)	4
MAXX (right side value)	8
MAXY (top side value)	12

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**

enabled characteristics mask	0
disabled characteristics mask	4
keyclick volume	8
screen saver timeout value	12

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, defined by the \$QVBDEF macro, consist of the following bits:	
Characteristic	Default	Meaning
QV\$M_KEY_AUTORPT	On	Key held down automatically repeats.
QV\$M_KEY_KEYCLICK	On	Keyclick sounds on each keystroke.
QV\$M_KEY_UDF6	Off	Function keys F6 through F10 generate up/down transitions.
QV\$M_KEY_UDF11	Off	Function keys F11 through F14 generate up/down transitions.
QV\$M_KEY_UDF17	Off	Function keys F17 through F20 generate up/down transitions.
QV\$M_KEY_UDHELPDO	Off	Function keys HELP and DO generate up/down transitions.

QVSS/QDSS Data Structures

Field	Use
	<p>QV\$M_KEY_UDE1 Off Function keys E1 through E6 generate up/down transitions.</p> <p>QV\$M_KEY_UDARROW Off Arrow keys generate up/down transitions.</p> <p>QV\$M_KEY_UDNUMKEY Off Numeric keypad keys generate up/down transitions.</p> <p>QV\$M_SYS_SCRSAV On 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.</p>
Keyclick volume	Must be a value from 1 (loudest) to 8 (softest). Default is 3.
Screen saver timeout value	Represents minutes of inactivity that must elapse before the screen saver is activated. The value must be from 1 to 1440. If a value of 0 is specified, the timeout value is not changed. Default is 15.

B

QDSS-Specific Data Structures

This appendix illustrates and describes the QDSS data structures 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 at system installation time.)

VWSSYSDEF defines data type structures and constants, including offset values that define each field in the structure. 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 the structure.

To use any predefined constants and offsets, "include" or "insert" the SYSLIBRARY:VWSSYSDEF file in every module where you reference it. Become familiar with the way VWSSYSDEF defines the DOP structure for the programming language you are using.

dop queue structure (req_structure)

REQ\$L_REQUEST_FLINK		0
REQ\$L_REQUEST_BLINK		4
REQ\$L_RETURN_FLINK		8
REQ\$L_RETURN_BLINK		12
REQ\$L_RETURN_LARGE_FLINK		16
REQ\$L_RETURN_LARGE_BLINK		20
REQ\$W_LARGE_DOP_SIZE	REQ\$W_SMALL_DOP_SIZE	24
REQ\$L_APPLICATION_RESERVED		28

QDSS-Specific Data Structures

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.

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)		120
QDB\$L_SYSVP		124
QDB\$W_ON_SCREEN_Y	QDB\$W_ON_SCREEN_X	128
QDB\$W_ON_SCREEN_HEIGHT	QDB\$W_ON_SCREEN_WIDTH	132
QDB\$W_SCROLL_Y	QDB\$W_SCROLL_X	136
QDB\$W_SCROLL_HEIGHT	QDB\$W_SCROLL_WIDTH	140
QDB\$W_FREE_1_Y	QDB\$W_FREE_1_X	144
QDB\$W_FREE_1_HEIGHT	QDB\$W_FREE_1_WIDTH	148
QDB\$W_FREE_2_Y	QDB\$W_FREE_2_X	152
QDB\$W_FREE_2_HEIGHT	QDB\$W_FREE_2_WIDTH	156
QDB\$W_FREE_3_Y	QDB\$W_FREE_3_X	160
QDB\$W_FREE_3_HEIGHT	QDB\$W_FREE_3_WIDTH	164
QDB\$W_FONT_Y	QDB\$W_FONT_X	168
QDB\$W_FONT_HEIGHT	QDB\$W_FONT_WIDTH	172
QDB\$W_CLIP_SAVE_Y	QDB\$W_CLIP_SAVE_X	176
QDB\$W_CLIP_SAVE_HEIGHT	QDB\$W_CLIP_SAVE_WIDTH	180
QDB\$L_COLOR_INDICES		184
QDB\$L_COLOR_COLORS		188
QDB\$L_COLOR_RBITS		192
QDB\$L_COLOR_GBITS		196
QDB\$L_COLOR_BBITS		200
QDB\$L_COLOR_IBITS		204
QDB\$L_COLOR_RES_INDICES		208
QDB\$L_COLOR_REGEN		212

(Continued on next page)

QDSS-Specific Data Structures

Example 5-5 (Cont.) Calling Program for Example Subroutines

QDB\$L_COLOR_MAPS	216
QDB\$L_COLOR_INTENSITY_FLAG	220

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_Y	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 writable memory.
QDB\$W_FREE_1_Y	Y coordinate of the lower left-hand corner of writable memory.
QDB\$W_FREE_1_WIDTH	Width of writable memory.
QDB\$W_FREE_1_HEIGHT	Height of writable memory.
QDB\$W_FONT_X	X coordinate of the lower left-hand corner of bitmap storage area.
QDB\$W_FONT_Y	Y 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.

QDSS-Specific Data Structures

Field	Use
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	Number of hardware color maps (always 1 for QDSS).
QDB\$L_COLOR_INTENSITY_FLAG	Indicates color (0) or intensity (1) mode.

The following constants are defined in conjunction with the QDB.

Constant	Value
QVB\$C_LENGTH	Length of the QVB structure.

return queue structure (ret_structure)

RET\$L_RETURN_FLINK	0	
RET\$L_RETURN_BLINK	4	
RET\$L_RETURN_LARGE_FLINK	8	
RET\$L_RETURN_LARGE_BLINK	12	
RET\$W_LARGE_DOP_SIZE	RET\$W_SMALL_DOP_SIZE	16
RET\$L_APPLICATION_RESERVED	20	

QDSS-Specific Data Structures

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	Longword reserved for use by the application.

transfer parameter block (tpb) (tpb_structure)

TPB\$W_X_SOURCE	TPB\$B_SIZE	TPB\$B_TYPE	0
TPB\$W_WIDTH	TPB\$W_Y_SOURCE		4
TPB\$W_X_TARGET	TPB\$W_HEIGHT		8
TPB\$W_X_TARGET_VEC1	TPB\$W_Y_TARGET		12
TPB\$W_L_TARGET_VEC1	TPB\$W_Y_TARGET_VEC1		16
TPB\$W_Y_TARGET_VEC2	TPB\$W_X_TARGET_VEC2		20
	TPB\$W_L_TARGET_VEC2		

QDSS-Specific Data Structures

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 later 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.

QDSS-Specific Data Structures

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	0
URD\$W_Y_MAX	URD\$W_X_MAX	4
URD\$W_Y_BASE	URD\$W_X_BASE	8

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.

QDSS-Specific Data Structures

The following constants are defined in conjunction with the URD.

Constant	Value
URD\$C_ LENGTH	Length of the structure.

C

QDSS Writing Modes

This appendix lists all QDSS writing modes and describes the logical operations that can be performed when two graphic objects intersect. This operation occurs among the *destination*, the bits composing 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 foreground and background color to determine bit settings for the pixels that compose the intersecting area.

The writing mode names are acronyms that reflect the function performed. To interpret (and remember) the names, read 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 lists QDSS writing modes and their functions.

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 ORed 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.

QDSS Writing Modes

In addition, you can 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 DOP bitmap ID field 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 DOP bitmap ID field 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.

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 D.1.2 shows how a typical program might be designed for the QVSS driver. This simple program uses most of the QIO functions available to the QVSS driver to perform 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 and a new default system cursor pattern.
- 3 Sets up two separate cursor regions on the screen.
 - a. Block-shaped cursor is located in the lower left corner.
 - b. Cross-shaped cursor is located in the upper right corner.
- 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 up two keyboard regions.
 - a. 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 is a "C," issues a cycle QIO on the keyboard list, which activates the other keyboard region; then delivers a control AST for the new region.
 - If the character "F" is typed, sets event flag 2, which terminates the program.
 - c. The control AST specified for each region sends a message to the terminal, indicating that a control AST was delivered.

QVSS Programming Example

- 9 Enables an AST region for pointer buttons.
 - a. Sets up an AST to be called each time a pointer button is pressed/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 issues a cycle QIO on the keyboard list.
 - The cycle QIO delivers the control AST for the new keyboard region.
- 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. Enables the AST routine to 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
```


QVSS Programming Example

```

;
; 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:
    .BLKL 1                        ; State of buttons
MOUSE_XY:
    .BLKL 1                        ; Current pointer X,Y coordinates
CHARACTER:
    .BLKL 1                        ; Keyboard character
DESC:
    .LONG 2
    .LONG CHARACTER
IOSB_BLOCK:
    .QUAD 0                        ; IOSB descriptor
DESC1:
    .LONG 1
    .LONG IOSB_BLOCK+4
    .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

```

QVSS Programming Example

```

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

```

QVSS Programming Example

```

.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
; pointer cursor used in call
    .WORD   ^b0000001110000000
    .WORD   ^b0000011000000000
    .WORD   ^b0000001100000000
    .WORD   ^b0000001100000000
    .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

```

QVSS Programming Example

```

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 ^X0940
.WORD ^X09c0
.WORD ^X0840
.WORD ^X0840
.WORD ^X0fc0

SPRAYCAN_HS:                                 ; Spraycan hot spot definition
.LONG 0
.LONG 1
.SBTL 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$: $ASSIGN_S DEVNAM=WS_DEVNAM,-             ; Assign another cursor channel using
        CHAN=CUR_CHAN2                       ; logical name and channel number
    BLBS R0,20$                               ; Check for success
    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

```

QVSS Programming Example

```

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

QV$CURSOR2:                             ; 16 * 16 Cursor pattern 2 (cross)
    .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
    .WORD ^b0000011110000000
    .SBTTL SET_ONESHOT - SET UP ONE-SHOT QIO

; ++
;
; FUNCTIONAL DESCRIPTION:
;
; Set a 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:
    $ASSIGN_S DEVNAM=WS_DEVNAM,-        ; Assign a keyboard channel using
    CHAN=KBD_CHAN1                    ; logical name and channel number
    BLBS RO,10$                        ; Check for success
    BRW ERROR

```

QVSS Programming Example

```

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

```

QVSS Programming Example

```

;
; 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

```


QVSS Programming Example

```

.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    RO,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>

```

QVSS Programming Example

```

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:
    $ASSIGN_S DEVNAM=WS_DEVNAM,- ; Assign keyboard channel 2 using
                CHAN=KBD_CHAN2 ; logical name and channel number
    BLBS RO,10$ ; 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 RO,20$ ; Check for success
    BRW ERROR

```

QVSS Programming Example

```

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                       ; Keyboard AST address
      .LONG  ACK2                           ; AST parameter
      .LONG  0                               ; AST delivery mode
      .LONG  CHARACTER                       ; Input token
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

```

QVSS Programming Example

```

        .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

```

QVSS Programming Example

```

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

```

QVSS Programming Example

```
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
```

QVSS Programming Example

```

        .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

```

QVSS Programming Example

```
; 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
```


E

Keyboard Table Macro

This appendix contains the macro used to generate a keyboard table.

```
.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
```

Keyboard Table Macro

```
.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>
```

Keyboard Table Macro

```

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

```


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
;
; .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
```

Compose Table Macros

```
.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
```

G

Default Three-Stroke Compose Table 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$COMPOSE_KEY <^a/" />,<^a/E />,,<^xch>
VC$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>
```

Default Three-Stroke Compose Table Values

```

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>

```


Default Three-Stroke Compose Table Values

```

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$COMPOSE_KEY <^a/O/,>,<^a/E />,,<^xd7>
VC$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/O />,,<^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>

```

; order sensitive

Default Three-Stroke Compose Table Values

```

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$COMPOSE_KEY <^a/a/>,<^a/a/>,<^e>
VC$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$COMPOSE_KEY <^a/c/>,<^a/O/>,,<^xa9>
VC$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>

```

Default Three-Stroke Compose Table Values

```

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$COMPOSE_KEY <^a/x/>,<^a/0/>,,<^xa8>
VC$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>

;
; 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

```


H

\$QIO System Service Description

The \$QIO system 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.

\$QIO System Service
\$QIO System Service Description

\$QIO System Service Description

\$QIO SYSTEM SERVICE

FORMAT **SYSSQIO** *[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 sets 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**

\$QIO System Service

\$QIO System Service Description

Device-specific function codes and function modifiers specifying the operation to be performed. The **func** is a longword value containing the function code.

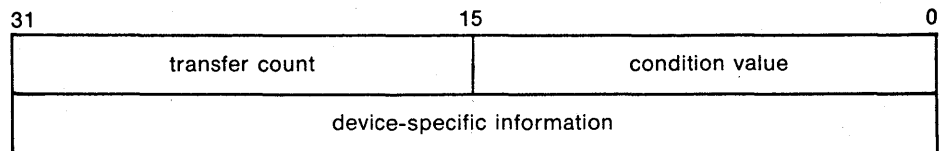
Each device has its own function codes and function modifiers. Refer to Chapter 3 and Chapter 4 for complete information about the function codes and function modifiers that apply to the particular I/O operation you want to perform.

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 shows 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.

Although 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 values returned in R0 and the I/O status block provide information about different aspects of the call to the \$QIO service:
 - The condition value returned in R0 provides information about the success or failure of the service call itself.

\$QIO System Service

\$QIO System Service Description

- The condition value returned in the status provides information about the success or failure of the service operation.

Therefore, to assess accurately 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

\$QIO System Service

\$QIO System Service Description

To synchronize completion for \$QIO, perform either of the following procedures.

- Specify the `astadr` argument with an AST routine execute when the I/O completes.
- Call the Synchronize (\$SYNCH) service to await completion of the I/O operation. (Because \$QIOW completes synchronously, this is the better choice when you require synchronous completion.)

`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

<code>SS\$_NORMAL</code>	Service successfully completed. The I/O request was successfully queued.
<code>SS\$_ABORT</code>	A network logical link was broken.
<code>SS\$_ACCVIO</code>	Either the I/O status block cannot be written by the caller, or the parameters for device-dependent function codes are incorrectly specified.
<code>SS\$_DEVOFFLINE</code>	The specified device is offline and not currently available for use.
<code>SS\$_EXQUOTA</code>	The process has done any of the following: <ol style="list-style-type: none"> 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 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
<code>SS\$_ILLEFC</code>	An illegal event flag number was specified.
<code>SS\$_INSFMEM</code>	Insufficient system dynamic memory is available to complete the service.
<code>SS\$_IVCHAN</code>	An invalid channel number was specified, that is, a channel number of 0 or a number larger than the number of channels available.
<code>SS\$_NOPRIV</code>	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.
<code>SS\$_UNASEFC</code>	The process is not associated with the cluster containing the specified event flag.

\$QIO System Service

\$QIO System Service Description

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, probably because of 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 is invalid at the remote node.
SS\$_IVDEVNAM	The NCB has an invalid format or content.
SS\$_LINKEXIT	The network partner task 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 probably because of 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.

DEC Multinational Character Set

Figure I-1 represents the ASCII character set (characters with decimal values 0 through 127), the first half of the DEC multinational character set.

The first half of each numbered column 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 second half of each column identifies the character by the binary value of the byte; the value is stated in three radices—octal, decimal, and hexadecimal. For example, under ASCII conventions, the letter uppercase A has a storage value of hexadecimal 41 (a bit configuration of 01000001), equivalent to 101 in octal notation and 65 in decimal notation.

Figure I-2 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).

DEC Multinational Character Set

Figure I-1 DEC Multinational Character Set—I

ROW	COLUMN							
	0	1	2	3	4	5	6	7
BITS								
b8 b7 b6 b5 b4 b3 b2 b1								
0	0 0 0 0	NUL	DLE	SP	0	@	P	p
1	0 0 0 1	SOH	DC1 (XON)	!	1	A	Q	a q
2	0 0 1 0	STX	DC2	"	2	B	R	b r
3	0 0 1 1	ETX	DC3 (XOFF)	#	3	C	S	c s
4	0 1 0 0	EOT	DC4	\$	4	D	T	d t
5	0 1 0 1	ENQ	NAK	%	5	E	U	e u
6	0 1 1 0	ACK	SYN	&	6	F	V	f v
7	0 1 1 1	BEL	ETB	'	7	G	W	g w
8	1 0 0 0	BS	CAN	(8	H	X	h x
9	1 0 0 1	HT	EM)	9	I	Y	i y
10	1 0 1 0	LF	SUB	*	:	J	Z	j z
11	1 0 1 1	VT	ESC	+	;	K	[k {
12	1 1 0 0	FF	FS	,	<	L	\	l
13	1 1 0 1	CR	GS	-	=	M]	m }
14	1 1 1 0	SO	RS	.	>	N	^	n ~
15	1 1 1 1	SI	US	/	?	O	_	o DEL

KEY

CHARACTER	ESC	33	OCTAL
		27	DECIMAL
		1B	HEX

ZK-1752-84

Figure I-2 DEC Multinational Character Set—II

8		9		10		11		12		13		14		15		COLUMN	ROW
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		118 117 116 115 114 113 112 111	BITS
200 128 80	DCS	220 144 90		240 160 A0	°	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	i	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	e	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	
204 132 84	IND	224 148 94		244 164 A4		264 180 B4	Ä	304 196 C4	Ö	324 212 D4	ä	344 228 E4	ö	364 244 F4	0 1 0 0	4	
205 133 85	NEL	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	
206 134 86	SSA	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	
207 135 87	ESA	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	
210 136 88	HTS	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	
211 137 89	HTJ	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	
212 138 8A	VTS	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	
213 139 8B	PLD	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	
214 140 8C	PLU	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	
215 141 8D	RI	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	
216 142 8E	SS2	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	
217 143 8F	SS3	237 159 9F		257 175 AF	¿	277 191 BF	Ï	317 207 CF	ß	337 223 DF	ï			377 255 FF	1 1 1 1	15	

KEY

CHARACTER	ESC	33	OCTAL
		27	DECIMAL
		1B	HEX

ZK-1753-84

J

ISO Latin Nr 1 Supplemental Character Set

The ASCII character set (characters with decimal values 0 through 127) comprises the first half of the ISO Latin character set. 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.

The second half of the ISO Latin character set is represented as characters with decimal values 128 through 255.

NOTE: Characters in the first half of the ISO Latin character set can be output on a VT200 or VT100 series terminal. Characters in the second half of the ISO Latin character set cannot be output on a VT100 series terminal.

Index

A

- Absolute device coordinates
 - See ADC
 - ADC (absolute device coordinates) • 1-15, 2-26, 2-32
 - Alternate windowing system • 2-24
-

B

- Background color index • 5-19
 - Bitmap
 - accessing • 1-9, 1-11
 - exclusive access to • 2-32
 - loading into offscreen memory • 2-29
 - loading into storage area • 1-11
 - reading • 2-28
 - storage area • 1-11
 - systemwide • 4-17
 - transferring • 1-11
 - writing • 2-28
 - Bitmap ID • 2-29
 - Bitmap_glyphs field • 5-21
 - Bitmap_ID field • 5-21
 - Button simulation block • A-1
 - Button transition value • 2-20
-

C

- Channel
 - assigning • 2-2
 - assigning to viewport • 2-25
 - deassigning • 2-41
- Character
 - composing nonstandard • 2-11
 - input • 2-20
- Color
 - displaying • 2-46
 - identifying system type • 2-47
- Color map
 - changing values in • 2-47
- Compose sequence • 2-11

- Compose sequence (cont'd.)
 - three-stroke • 2-11
 - two-stroke • 2-11
 - Compose sequence table • 2-12
 - constructing • 2-12
 - default • 2-16
 - initializing • 2-15
 - loading • 2-15, 2-16
 - macro to generate • F-1
 - terminating • 2-15
 - three-stroke • 2-12
 - two-stroke • 2-13
 - Control AST • 2-3
 - Cursor hot spot structure • A-1
 - Cursor pattern
 - defining • 2-21
 - multiplane • 2-22, 3-6
 - Cursor pattern entry
 - creating • 2-21
 - Cursor pattern list • 1-12
 - Cycling operation • 1-14
-

D

- Data rectangle values block • A-2
- Data structure
 - using predefined structure • B-1
- DEC multinational character set • I-1
- Deferred queue • 1-18
 - deleting • 2-46
 - executing • 2-46
- Define Pointer Cursor Pattern QIO • 1-12, 2-21, 3-3
- Define Viewport Region QIO • 2-26, 2-31, 2-33, 2-46, 4-3
- Delete Bitmap DOP • 5-23
- Delete Deferred Queue Operation QIO • 2-46, 4-5
- Diacritical mark • 2-9, 2-11, 2-13
- Diacritical table • 2-13
- DOP (drawing operation primitive) • 1-6, 5-1 to 5-81
 - allocating memory for • 1-17
 - allocating storage for • 5-3
 - drawing with • 2-28
 - entering on request queue • 1-17
 - executing • 5-5

Index

DOP (drawing operation primitive)
 executing (cont'd.)
 asynchronously • 5-5
 initializing • 5-9 to 5-14
 using predefined structure • 5-12
 large • 5-3
 small • 5-3
 state of • 2-29
 structuring • 5-9 to 5-14
 using predefined structure • 5-12
DOP Common block • 5-2, 5-10, 5-16
 initializing • 5-18 to 5-22
DOP queue structure • B-1
DOP Unique block • 5-2, 5-12
DOP Variable block • 5-2, 5-12
Dop_line_array predefined structure • 5-41
Dop_move_array predefined structure • 5-52, 5-63
Dop_move_r_array predefined structure • 5-24,
 5-56
Dop_point_array predefined structure • 5-44
Dop_poly_array predefined structure • 5-48
Dop_structure predefined structure • 5-16
Dop_vtext_array predefined structure • 5-37
Draw Complex Line DOP • 5-24
Draw Fixed Text DOP • 5-28
Drawing operation primitive
 See DOP
Drawing to the QVSS screen • 2-24
Draw Lines DOP • 5-31
Draw Points DOP • 5-34
Draw Variable Text DOP • 5-37
Driver differences
 alternate windowing • 1-3
 bitmap manipulation • 1-3
 color • 1-3
 driver interface • 1-3
 hardware interface • 1-4
 UISDC interface • 1-3
 UIS interface • 1-3

E

Enable Button Transition QIO • 1-13, 2-18, 3-8
Enable Data Digitizing QIO • 3-13
Enable Function Keys QIO • 3-17
Enable Input Simulation QIO • 3-20
Enable Keyboard Input QIO • 1-14, 2-3, 2-6, 3-23
Enable Keyboard Sound QIO • 3-29
Enable Pointer Movement QIO • 1-13, 2-17, 3-31
Enable User Entry QIO • 3-35

Index-2

Events • 1-4
Event tracking • 1-12
Example
 assign channel • 2-4
 AST routine • 2-4
 keyboard request • 2-4
 QVSS sample program • D-1
Execute Deferred Queue QIO • 1-18, 2-46, 4-6
Exit AST • 2-18

F

Fill Lines DOP • 5-41
Fill Point DOP • 5-44
Fill Polygon DOP • 2-41, 5-48
Flags field • 5-22
Foreground color index • 5-19
Free DOP • 1-17
Free_1 area • 1-10, 2-31

G

Get Color Map Entries QIO • 2-48, 4-7
Get Free DOPs QIO • 1-17, 4-9
Get Keyboard Characteristics QIO • 1-6, 3-37
Get Next Input Token QIO • 2-3, 2-21, 3-39
Get Number of List Entries QIO • 3-40
Get System Information QIO • 1-11, 2-1, 2-41,
 3-41
Get Viewport ID QIO • 1-16, 2-25, 4-10

H

Hardware color map • 2-47
 determining current values of • 2-48
 loading values into • 2-48
Hardware cursor • 1-8
Hardware look-up table • 2-47
Hold Viewport Activity QIO • 2-30, 4-12

I

Image
 drawing • 2-29

- Image (cont'd.)
 - storing • 2-29
 - Initialize Screen QIO • 2-1, 3-42
 - Input
 - intercepting • 2-21
 - Insert DOP QIO • 1-17, 2-30, 2-41, 4-13
 - INSQUE (Insert Entry in Queue) instruction • 5-9
 - Intensity value • 2-47
 - Intercepting input • 2-21
 - IO\$C_QD_COLOR_CHAR function code • 4-23
 - IO\$C_QD_DELETE_DEFERRED function code • 4-5
 - IO\$C_QD_EXECUTE_DEFERRED function code • 4-6
 - IO\$C_QD_GET_COLOR function code • 4-7
 - IO\$C_QD_GET_FREE_DOPS function code • 4-9
 - IO\$C_QD_GET_VIEWPORT_ID function code • 4-10
 - IO\$C_QD_HOLD function code • 4-12, 4-18
 - IO\$C_QD_LOAD_BITMAP function code • 4-14
 - IO\$C_QD_NOHOLD function code • 4-22
 - IO\$C_QD_OCCLUDED_SUSPEND function code • 4-28
 - IO\$C_QD_RESUME_VP function code • 4-23
 - IO\$C_QD_SETCOLOR function code • 4-24
 - IO\$C_QD_SET_VIEWPORT_REGIONS function code • 4-3
 - IO\$C_QD_START function code • 4-26
 - IO\$C_QD_STOP function code • 4-27
 - IO\$C_QD_SUSPEND_VP function code • 4-29
 - IO\$C_QV_ENABLE_DIGITIZING function code • 3-13
 - IO\$C_QV_ENABUTTON function code • 3-8
 - IO\$C_QV_ENAFNKEY function code • 3-17
 - IO\$C_QV_ENAKB function code • 3-23
 - IO\$C_QV_ENAUSER function code • 3-35
 - IO\$C_QV_GETKB_INFO function code • 3-37
 - IO\$C_QV_GETSYS function code • 3-41
 - IO\$C_QV_GET_ENTRIES function code • 3-40
 - IO\$C_QV_INITIALIZE • 3-42
 - IO\$C_QV_LOAD_COMPOSE_TABLE function code • 3-43
 - IO\$C_QV_LOAD_KEY_TABLE function code • 3-45
 - IO\$C_QV_MODIFYKB function code • 3-47
 - IO\$C_QV_MODIFYSYS function code • 3-51
 - IO\$C_QV_MOUSEMOV function code • 3-31
 - IO\$C_QV_SETCURSOR function code • 3-3
 - IO\$C_QV_SIMULATE function code • 3-20
 - IO\$C_QV_SOUND function code • 3-29
 - IO\$C_QV_USE_DEFAULT_TABLE function code • 3-55, 3-56
 - IO\$M_QD_INTENSITY function modifier • 4-7, 4-24
 - IO\$M_QD_RESERVED_COLORS function modifier • 4-7, 4-24
 - IO\$M_QD_SYSTEM_WIDE function modifier • 4-14
 - IO\$M_QV_ACTIVE function modifier • 3-47
 - IO\$M_QV_BIND function modifier • 3-3
 - IO\$M_QV_COMPOSE2 function modifier • 3-55
 - IO\$M_QV_COMPOSE3 function modifier • 3-55
 - IO\$M_QV_CYCLE function modifier • 3-23, 3-35
 - IO\$M_QV_DELETE function modifier • 3-3, 3-8, 3-13, 3-23, 3-31, 3-35
 - IO\$M_QV_KEYS function modifier • 3-56
 - IO\$M_QV_LAST function modifier • 3-3, 3-8, 3-23, 3-31, 3-35
 - IO\$M_QV_LOAD_DEFAULT function modifier • 3-3, 3-43, 3-45
 - IO\$M_QV_PURG_TAH function modifier • 3-8, 3-23
 - IO\$QV_TWO_PLANE_CURSOR function modifier • 3-3
 - IO\$QV_USE_DEFAULT function modifier • 3-3
 - IO\$SENSEMODE • 1-5
 - IO\$SETMODE • 1-5
 - ISO Latin Nr 1 supplemental character set • J-1
 - item_type field • 5-18
-
- ## K
-
- Key
 - defining • 2-7
 - Keyboard
 - activating • 1-14
 - cycling • 1-14
 - popping • 1-14
 - programming • 2-7
 - receiving input from • 2-3
 - using • 2-3 to 2-16
 - virtual • 2-4
 - Keyboard characteristics • 2-6
 - defining • 2-6
 - Keyboard characteristics block • 2-4, A-3
 - Keyboard entry list • 1-14, 2-3
 - Keyboard request AST specification block • A-2
 - Keyboard table
 - constructing • 2-9
 - initializing • 2-7
 - loading • 2-7, 2-11
 - macro to generate • E-1
 - modifying • 2-7

Index

Keyboard table (cont'd.)

terminating • 2-9

Key-click volume • 2-6

Keystroke AST specification block • A-5

L

LIB\$GET_VM • 5-7

List entry • 1-5, 1-12

Load Bitmap QIO • 1-11, 2-29, 4-14

Load compose sequence table • 2-16

Load Compose Sequence Table QIO • 2-12, 2-16, 3-43

Load Keyboard Table QIO • 2-7, 2-11, 3-45

M

Macro

compose table • F-1

keyboard table • E-1

Mapping • 2-24.1

Memory • 2-24.1

offscreen • 1-7, 1-10, 1-16

onscreen • 1-7, 1-9, 1-16

Memory usage • 1-6

Meta-key • 2-19, 3-9, 3-15, 3-24

Modify Keyboard Characteristics QIO • 2-7, 3-47

Modify Systemwide Characteristics QIO • 2-6, 3-51

Mouse • 2-18

Move/Rotate Area DOP • 5-56

Move Area DOP • 5-52

N

New cursor position structure • A-8

New pointer position structure • A-8

Notify Deferred Queue Full QIO • 1-18, 2-46, 4-18

O

Occlusion • 1-15, 1-18

handling • 2-31

handling with update regions • 1-16

Offscreen memory • 1-7, 1-10, 1-16, 2-31

Onscreen memory • 1-7, 1-9, 1-16

Oppcount field • 5-18

P

Plot_args predefined structure • 5-31, 5-34, 5-41, 5-44, 5-48

Pointer • 2-18

using • 2-17

Pointer button characteristics block • A-6

Pointer button transition • 2-17

creating entry • 2-18

Pointer button transition list • 1-13

Pointer motion AST specification block • A-7

Pointer movement • 2-17

creating entry • 2-17

Pointer movement list • 1-13

Pointer movement list entry • 2-18

Pointer position • 2-20

Popping operation • 1-14

Puck • 2-18

Q

QDB (QDSS block) structure • B-3

QDSS block

See QDB

QDSS Viewport • 2-25

QIO interface • 1-5

Queue manipulation • 1-6

QVB\$_MAIN_VIDEOADDR • 2-24, 2-24.1

QVB\$_VIDEOADDR • 2-24.1

QVB (QVSS block) structure • A-10

\$QVBDEF.MLB • 2-7

Qvb_common_structure • A-10

Qvb_qdss_structure predefined structure • B-3

QV-DELETE • 2-4, 2-18

QV-LAST • 2-17

QVSS block

See QVB

QVSS control driver

sample program • D-1

QV_DELETE • 2-19, 2-22

QV_LAST • 2-19, 2-22

QV_PURG_TAH • 2-19

R

Read Bitmap QIO • 1-11, 2-29, 2-33, 2-45, 4-19
 Region • 1-4
 defining • 1-5
 placing on deferred queue • 2-45
 Region descriptor • 2-18
 Release Hold QIO • 2-30, 4-22
 Request AST • 2-3
 Request queue • 1-6, 1-16, 5-6
 Req_structure predefined structure • B-1
 Reserved function keystroke AST specification block • A-14
 Resume Request Queue QIO • 2-30
 Resume Viewport Activity DOP • 2-30, 5-61
 Resume Viewport Activity QIO • 4-23
 Return queue • 1-17, 5-4
 alternate • 1-18
 Return queue structure • B-5
 Ret_structure predefined structure • B-5
 Revert to Default Compose Table QIO • 2-16, 3-55
 Revert to Default Keyboard Table QIO • 2-11, 3-56

S

Scanline • 2-24.1
 Scanline map • 1-7, 1-8, 2-2, 2-24.1
 obtaining address of base • 2-24.1
 Screen
 drawing to • 2-24
 initialization • 2-1
 mapping video memory to • 2-24.1
 writing to • 1-6
 Screen event
 tracking • 1-12
 Screen rectangle values block • A-14
 Screen saver time • 2-6
 Scroll area • 1-10
 Scroll Area DOP • 5-63
 Scrolling save area • 1-10
 Set Color Characteristics QIO • 2-47, 4-23
 Set Color Map Entries QIO • 2-47, 2-48, 4-24
 Set Viewport Region QIO • 1-15
 Source index • 5-19
 Start Request Queue DOP • 5-67
 Start Request Queue QIO • 2-26, 2-30, 2-33, 4-26
 Start Viewport Activity DOP • 2-30
 Stop Request Queue DOP • 5-69

Stop Request Queue QIO • 2-30, 2-32, 4-27
 Stop Viewport Activity DOP • 2-30, 2-41
 Stop_args predefined structure • 5-61, 5-67, 5-69, 5-70
 Stylus • 2-18
 Suspend Occluded Viewport Activity QIO • 4-28
 Suspend Request Queue QIO • 2-30
 Suspend Viewport Activity DOP • 2-30, 5-70
 Suspend Viewport Activity QIO • 4-29
 SYS\$ASSIGN • 2-2
 SYS\$DASSGN • 2-41
 SYS\$QIO • H-2
 SYSTARTUP.COM • 2-24
 System characteristics block • 2-6, A-15
 System information block • 1-6, 2-1
 Systemwide viewport • 1-15, 2-25

T

Text_args predefined structure • 5-28, 5-37
 Token • 2-19, 3-9, 3-15
 TPB (transfer parameter block) • 2-33, B-6
 Tpb_structure predefined structure • B-6
 Transfer parameter block
 See TPB
 Type-ahead buffer
 getting input from • 2-20
 purging • 2-21
 using • 2-20

U

UIS\$CREATE_WINDOW • 5-6
 UIS\$WS_ALTAPPL • 2-24
 UISDC\$ALLOCATE_DOP • 5-3, 5-6, 5-12, 5-74
 UISDC\$EXECUTE_DOP_ASYNCH • 5-5, 5-78
 UISDC\$EXECUTE_DOP_SYNCH • 5-5, 5-80
 UISDC\$LOAD_BITMAP • 1-11, 5-76
 UISDC\$QUEUE_DOP • 5-5, 5-81
 Update region • 1-15
 and occlusion • 1-16
 defining • 2-25
 Update region definition
 See URD
 Update region definition block • B-8
 URD (update region definition) buffer • 2-25, 2-31
 Ur_d_structure predefined structure • B-8

Index

User-defined viewport • 1-15

V

VC\$COMPOSE_KEY • 2-15

VC\$COMPOSE_KEYEND • 2-15

VC\$COMPOSE_KEYINIT • 2-15

VC\$KEY • 2-7

VC\$KEYEND • 2-7

VC\$KEYINIT • 2-7

Video memory

 copying images to • 1-11

 drawing to • 1-11

 driver use of • 1-6

 mapping to screen • 2-24.1

 private • 2-24

 setting bits in • 2-24

Viewport • 1-4, 1-11, 1-15 to 1-16

 creating • 2-25

 defining • 2-25, 2-26

 deleting • 2-40

 erasing • 2-41

 moving • 2-45

 popping • 2-31, 2-36

 redefining • 2-31

 starting • 2-26

 synchronizing activity on • 2-29

Viewport ID • 1-16, 5-6

 getting • 2-25

Viewport-relative coordinates

 See VRC

Viewport request queue • 2-26

Viewport update regions • 1-15

VRC (viewport-relative coordinates) • 1-15, 2-26

\$VWSSYSDEF.MLB • 2-15

W

Window ID • 5-6

Windowing system

 alternate • 2-24

 enabling alternate • 2-24

Write Bitmap QIO • 1-11, 2-29, 2-33, 2-46, 4-30

Writing mode • 5-19

Writing mode field • 5-22

Z

Z-mode • 2-46, 2-47

Do Not Tear - Fold Here and Tape

digital™



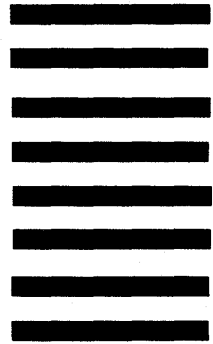
No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD, MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

VWS Engineering
Digital Equipment Corporation
110 Spitbrook Rd. ZKO3-2/S30
Nashua, New Hampshire 03062-2698



Do Not Tear - Fold Here